APPENDIX A

AMAZON.COM, INC.
EXHIBIT 1013
US 5,995,102

A research center for augmenting human intellect*

by DOUGLAS C. ENGELBART
and WILLIAM K. ENGLISH

Stanford Research Institute
Menlo Park, California

1 SUMMARY

1a This paper describes a multisponsor research center at Stanford Research Institute in man-computer interaction.

1a1 For its laboratory facility, the Center has a time-sharing computer (65K, 24-bit core) with a 4.5 megabyte swapping drum and a 96 megabyte file-storage disk. This serves twelve CRT work stations simultaneously.

1a1a Special hardware completely removes from the CPU the burden of display refreshing and input sampling, even though these are done directly out of and into core.

1a1b The display in a user's office appears on a high-resolution (875-line) commercial television monitor, and provides both character and vector portrayals. A relatively standard typewriter keyboard is supplemented by a five-key handset used (optionally) for entry of control codes and brief literals. An SRI cursor device called the "mouse" is used for screen pointing and selection.

1a1b1 The "mouse" is a hand-held X-Y transducer usable on any flat surface; it is described in greater detail further on.

1a2 Special-purpose high-level languages and associated compilers provide rapid, flexible development and modification of the repertoire of service functions and of their control procedures (the latter being the detailed user actions and computer feedback involved in controlling the application of these service functions).

1b User files are organized as hierarchical structures of data entities, each composed of arbitrary combinations of text and figures. A repertoire of coordinated service features enables a skilled user to compose, study, and modify these files with great speed and flexibility, and to have searches, analyses data manipulation, etc. executed. In particular, special sets of conventions, functions, and working methods have been developed to air programming, logical design, documentation, retrieval, project management, team interaction, and hard-copy production.

2 INTRODUCTION

2a In the Augmented Human Intellect (AHI) Research Center at Stanford Research Institute a group of researchers is developing an experimental laboratory around an interactive, multiconsole computer-display system, and is working to learn the principles by which interactive computer aids can augment their intellectual capability.

2b The research objective is to develop principles and techniques for designing an "augmentation system."

2b1 This includes concern not only for the technology of providing interactive computer service, but also for changes both in ways of conceptualizing, visualizing, and organizing working material, and in procedures and methods for working individually and cooperatively.

*Principal sponsors are: Advanced Research Projects Agency and National Aeronautics and Space Agency (NAS1–7897), and Rome Air Development Center F30602–68–C–0286.

2c The research approach is strongly empirical. At the workplace of each member of the subject group we aim to provide nearly full-time availability of a CRT work station, and then to work continuously to improve both the service available at the stations and the aggregate value derived therefrom by the group over the entire range of its roles and activities.

2d Thus the research group is also the subject group in the experiment.

2d1 Among the special activities of the group are the evolutionary development of a complex hardware-software system, the design of new task procedures for the system's users, and careful documentation of the evolving system designs and user procedures.

2d2 The group also has the usual activities of managing its activities, keeping up with outside developments, publishing reports, etc.

2d3 Hence, the particulars of the augmentation system evolving here will reflect the nature of these tasks—i.e., the system is aimed at augmenting a system-development project team. Though the primary research goal is to develop principles of analysis and design so as to understand how to augment human capability, choosing the researchers themselves as subjects yields as valuable secondary benefit a system tailored to help develop complex computer-based systems.

2e This "bootstrap" group has the interesting (recursive) assignment of developing tools and techniques to make it more effective at carrying out its assignment.

2e1 Its tangible product is a developing augmentation system to provide increased capability for developing and studying augmentation systems.

2e2 This system can hopefully be transferred, as a whole or by pieces of concept, principle and technique, to help others develop augmentation systems for aiding many other disciplines and activities.

2f In other words we are concentrating fully upon reaching the point where we can do all of our work on line—placing in computer store all of our specifications, plans, designs, programs, documentation, reports, memos, bibliography and reference notes, etc., and doing all of our scratch work, planning, designing, debugging, etc., and a good deal of our intercommunication, via the consoles.

2f1 We are trying to maximize the coverage of our documentation, using it as a dynamic and plastic structure that we continually develop and alter to represent the current state of our evolving goals, plans, progress, knowledge, designs, procedures, and data.

2g The display-computer system to support this experiment is just (at this writing) becoming operational. Its functional features serve a basic display-oriented user system that we have evolved over five years and through three other computers. Below are described the principal features of these systems.

3 THE USER SYSTEM

3a Basic Facility

3a1 As "seen" by the user, the basic facility has the following characteristics:

3a1a 12 CRT consoles, of which 10 are normally located in offices of AHI research staff.

3a1b The consoles are served by an SDS 940 time-sharing computer dedicated to full-time service for this staff, and each console may operate entirely independently of the others.

3a1c Each individual has private file space, and the group has community space, on a high-speed disc with a capacity of 96 million characters.

3a2 The system is not intended to serve a general community of time-sharing users, but is being shaped in its entire design toward the special needs of the "bootstrapping" experiment.

3b Work Stations

3b1 As noted above, each work station is equipped with a display, an alphanumeric keyboard, a mouse, and a five-key handset.

3b2 The display at each of the work stations (see Figure 1) is provided on a high-resolution, closed-circuit television monitor.

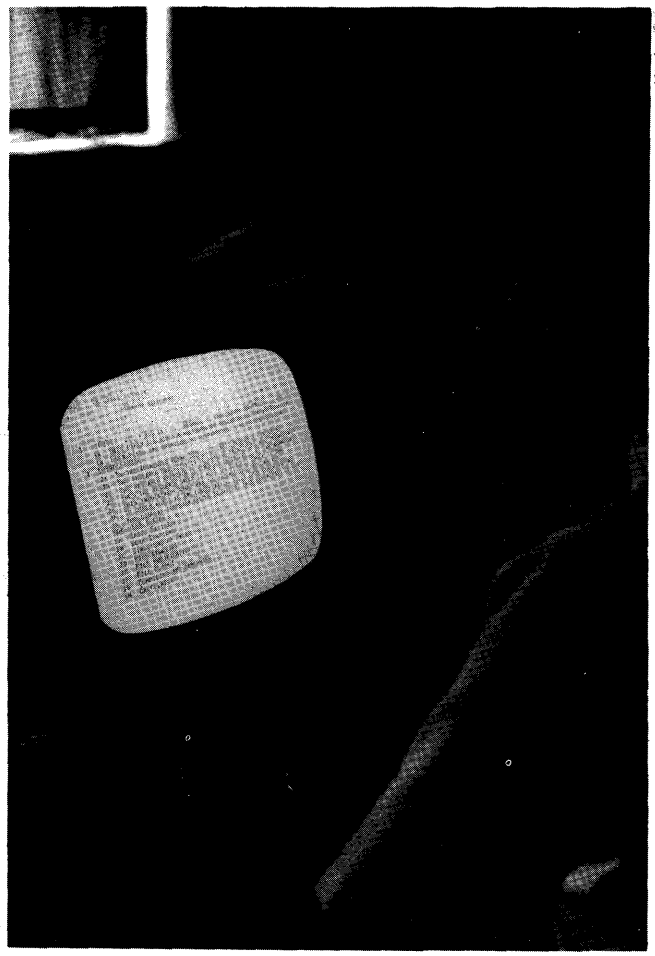


FIGURE 1—Typical work station, with TV display, typewriter keyboard, mouse, and chord handset

3b3 The alphanumeric keyboard is similar to a Teletype keyboard. It has 96 normal characters in two cases. A third-case shift key provides for future expansion, and two special keys are used for system control.

3b4 The mouse produces two analog voltages as het wo wheels (see Figure 2) rotate, each changing in proportion to the X or Y movement over the table top.

3b4a These voltages control—via an A/D converter, the computer's memory, and the display generator—the coordinates of a tracking spot with which the user may "point" to positions on the screen.

3b4b Three buttons on top of the mouse are used for special control.

3b4c A set of experiments, comparing (within our techniques of interaction) the

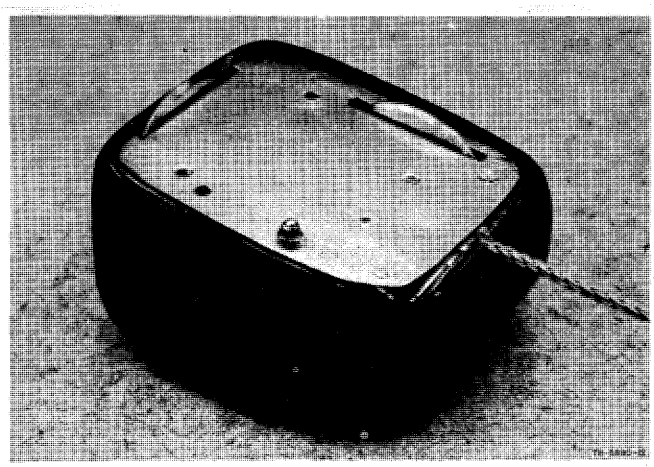


FIGURE 2—Underside of mouse

relative speed and accuracy obtained with this and other selection devices showed the mouse to be better than a light pen or a joystick (see Refs. English 1 and English 2).

3b4c1 Compared to a light pen, it is generally less awkward and fatiguing to use, and it has a decided advantage for use with raster-scan, write-through storage tube, projection, or multiviewer display systems.

3b5 The five-key handset has 31 chords or unique key-stroke combinations, in five "cases."

3b5a The first four cases contain lower- and upper-case letters and punctuation, digits, and special characters. (The chords for the letters correspond to the binary numbers from 1 to 26.)

3b5b The fifth case is "control case." A particular chord (the same chord in each case) will always transfer subsequent input-chord interpretations to control case.

3b5c In control case, one can "backspace" through recent input, specify underlining for subsequent input, transfer to another case, visit another case for one character or one word, etc.

3b5d One-handed typing with the handset is slower than two-handed typing with the standard keyboard. However, when the user works with one hand on the handset and one on the mouse, the coordinated in-

terspersion of control characters and short literal strings from one hand with mouse-control actions from the other yields considerable advantage in speed and smoothness of operation.

3b5d1 For literal strings longer than about ten characters, one tends to transfer from the handset to the normal keyboard.

3b5d2 Both from general experience and from specific experiment, it seems that enough handset skill to make its use worthwhile can generally be achieved with about five hours of practice. Beyond this, skill grows with usage.

3c **Structure of Files**

3c1 Our working information is organized into files, with flexible means for users to set up indices and directories, and to hop from file to file by display-selection or by typed-in file-name designations. Each file is highly structured in its internal organization.

3c1a The specific structure of a given file is determined by the user, and is an important part of his conceptual and "study-manipulate" treatment of the file.

3c2 The introduction of explicit "structuring" to our working information stems from a very basic feature of our conceptual framework (see Refs. Engelbart1 and Engelbart2) regarding means for augmenting human intellect.

3c2a With the view that the symbols one works with are supposed to represent a mapping of one's associated concepts, and further that one's concepts exist in a "network" of relationships as opposed to the essentially linear form of actual printed records, it was decided that the concept-manipulation aids derivable from real-time computer support could be appreciably enhanced by structuring conventions that would make explicit (for both the user and the computer) the various types of network relationships among concepts.

3c2b As an experiment with this concept, we adopted some years ago the convention of organizing all information into explicit hierarchical structures, with provisions for arbitrary cross-referencing among the elements of a hierarchy.

3c2b1 The principal manifestation of this hierarchical structure is the breaking up of text into arbitrary segments called "statements," each of which bears a number showing its serial location in the text and its "level" in an "outline" of the text. This paper is an example of hierarchical text structure.

3c2c To set up a reference link from Statement A to Statement B, one may refer in Statement A either to the location number of B or to the "name" of B. The difference is that the number is vulnerable to subsequent structural change, whereas the name stays with the statement through changes in the structure around it.

3c2c1 By convention, the first word of a statement is treated as the name of the statement, if it is enclosed in parentheses. For instance, Statement 0 on the screen of Figure 1 is named "FJCC."

3c2c2 References to these names may be embedded anywhere in other statements, for instance as "see(AFI)," where special format informs the viewer explicitly that this refers to a statement named "AFI," or merely as a string of characters in a context such that the viewer can infer the referencing.

3c2c3 This naminga nd linking, when added to the basic hierarchical form, yields a highly flexible general structuring capability. These structuring conventions are expected to evolve relatively rapidly as our research progresses.

3c3 For some material, the structured-statement form may be undesirable. In these cases, there are means for suppressing the special formatting in the final printout of the structured text.

3c4 The basic validity of the structured-text approach has been well established by our subsequent experience.

3c4a We have found that in both off-line and on-line computer aids, the concep-

tion, stipulation, and execution of significant manipulations are made much easier by the structuring conventions.

3c4b Also, in working on line at a CRT console, not only is manipulation made much easier and more powerful by the structure, but a user's ability to get about very quickly within his data, and to have special "views" of it generated to suit his need, are significantly aided by the structure.

3c4c We have come to write all of our documentation, notes, reports, and proposals according to these conventions, because of the resulting increase in our ability to study and manipulate them during composition, modification, and usage. Our programming systems also incorporate the conventions. We have found it to be fairly universal that after an initial period of negative reaction in reading explicitly structured material, one comes to prefer it to material printed in the normal form.

3d **File Studying**

3d1 The computer aids are used for two principal "studying" operations, both concerned with construction of the user's "views," i.e., the portion of his working text that he sees on the screen at a given moment.

3d1a Display Start

3d1a1 The first operation is finding a particular statement in the file (called the "display start"); the view will then begin with that statement. This is equivalent to finding the beginning of a particular passage in a hard-copy document.

3d1b Form of View

3d1b1 The second operation is the specification of a "form" of view—it may simply consist of a screenful of text which sequentially follows the point specified as the display start, or it may be constructed in other ways, frequently so as to give the effect of an outline.

3d1c In normal, off-line document studying, one often does the first type of operation, but the second is like a sissors-and-staple job and is rarely done just to aid one's studying.

3d1d (A third type of service operation that will undoubtedly be of significant aid to studying is question answering. We do not have this type of service.)

3d2 Specification of Display Start

3d2a The display start may be specified in several ways:

3d2a1 By direct selection of a statement which is on the display—the user simply points to any character in the statement, using the mouse.

3d2a2 If the desired display start is not on the display, it may be selected indirectly if it bears a "marker."

3d2a2a Markers are normally invisible. A marker has a name of up to five characters, and is attached to a character of the text. Referring to the marker by name (while holding down a special button) is exactly equivalent to pointing to the character with the mouse.

3d2a2b The control procedures make it extremely quick and easy to fix and call markers.

3d2a3 By furnishing either the name or the location number of the statement, which can be done in either of two basic ways:

3d2a3a Typing from the keyboard

3d2a3b Selecting an occurrence of the name or number in the text. This may be done either directly or via an indirect marker selection.

3d2b After identifying a statement by one of the above means, the user may request to be taken directly there for his next view. Alternately, he may request instead that he be taken to some statement bearing a specified structure relationship to the one specifically identified. For instance, when the user identifies Statement 3E4 by one of the above means (assume it to be a member of the list 3E1 through 3E7), he may ask to be taken to

3d2b1 Its successor, i.e., Statement 3E5

3d2b2 Its predecessor, i.e., Statement 3E3

3d2b3 Its list tail, i.e., Statement 3E7

3d2b4 Its list head, i.e., Statement 3E1

3d2b5 Its list source, i.e., Statement 3E

3d2b6 Its subhead, i.e., Statement 3E4A

3d2c Besides being taken to an explicitly identified statement, a user may ask to go to the first statement in the file (or the next after the current location) that contains a specified word or string of characters.

3d2c1 He may specify the search string by typing it in, by direct (mouse) selection, or by indirect (marker) selection.

3d3 Specification of Form of View

3d3a The "normal" view beginning at a given location is like a frame cut out from a long scroll upon which the hierarchical set of statements is printed in sequential order. Such a view is displayed in Figure 1.

3d3b Otherwise, three independently variable view-specification conditions may be applied to the construction of the displayed view: level clipping, line truncation, and content filtering. The view is simultaneously affected by all three of these.

3d3b1 Level: Given a specified level parameter, L (L = 1, 2, . . . , ALL), the view generator will display only those statements whose "depth" is less than or equal to L. (For example, Statement 3E4 is third level, 3E second, 4B2C1 fifth, etc.) Thus it is possible to see only first-level statements, or only first-, second-, and third level statements, for example.

3d3b2 Truncation: Given a specified truncation parameter, T (T = 1, 2, . . ., ALL), the view generator will show only the first T lines of each statement being displayed.

3d3b3 Content: Given a specification for desired content (written in a special high-level content-analysis language) the view generator optionally can be directed to display only those statements that have the specified content.

3d3b3a One can specify simple strings, or logical combinations thereof, or such things as having the word "memory" within four words of the word "allocation."

3d3b3b Content specifications are written as text, anywhere in the file. Thus the full power of the system may be used for composing and modifying them.

3d3b3c Any one content specification can then be chosen for application (by selecting it directly or indirectly). It is compiled immediately to produce a machine-code content-analysis routine, which is then ready to "filter" statements for the view generator.

3d3c In addition, the following format features of the display may be independently varied: indentation of statements according to level, suppression of location numbers and/or names of statements, and separation of statements by blank lines.

3d3d. The user controls these view specifications by means of brief, mnemonic character codes. A skilled user will readjust his view to suit immediate needs very quickly and frequently; for example, he may change level and truncation settings several times in as many seconds.

3d4 "Freezing" Statements

3d4a One may also pre-empt an arbitrary amount of the upper portion of the screen for holding a collection of "frozen" statements. The remaining lower portion is treated as a reduced-size scanning frame, and the view generator follows the same rules for filling it as described above.

3d4b The frozen statements may be independently chosen or dismissed, each may have line truncation independent of the rest, and the order in which they are displayed is arbitrary and readily changed. Any screen-select operand for any command may be selected from any portion of the display (including the frozen statements).

3d5 Examples

3d5a Figures 3 and 4 show views generated from the same starting point with different level-clipping parameters. This example happens to be of a program written in our Machine-Oriented language (MOL, see below).

3d5b Figure 5, demonstrates the freezing feature with a view of a program (the same one shown in Figure 8) written in our Control Metalanguage (CML, see below). Statements 3C, 3C2, 2B, 2B1, 2B2, 2B3, and 2B4 are frozen, and statements from 2J on are shown normally with L = 3, T = 1.

3d5b1 The freezing here was used to hold for simultaneous view four different functionally related process descriptions. The subroutines (+ BUG1SPEC) and (+ WAIT were located by use of the hop-to-name feature described above.

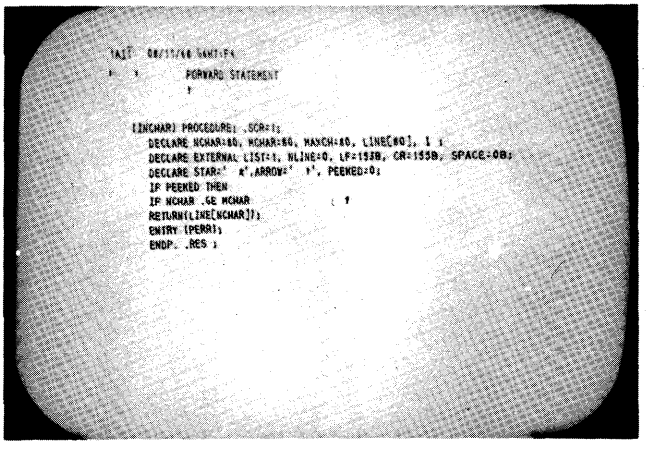


FIGURE 3—View of an MOL program, with level parameter set to 3 and truncation to 1

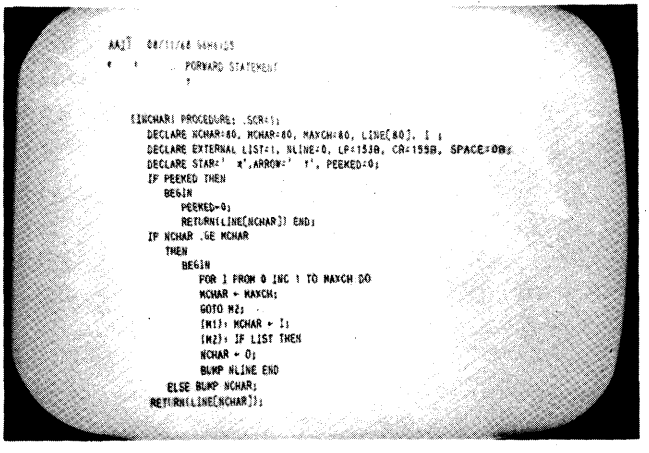


FIGURE 4—Same program as Figure 3, but with level parameter changed to 6 (several levels still remain hidden from view)

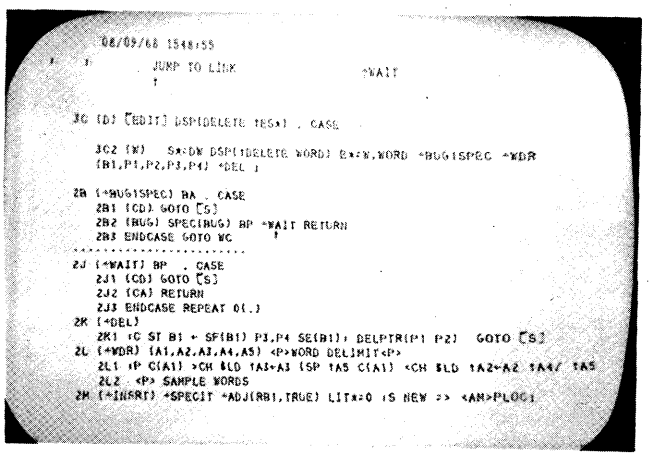


FIGURE 5—View of CML program, showing six frozen statements and illustrating use of reference hopping

3e **File Modification**

3e1 Here we use a standard set of editing operations, specifying with each operation a particular type of text entity.

3e1a Operations: Delete, Insert, Replace, Move, Copy.

3e1b Entities (within text of statements): Character, Text (arbitrary strings), Word, Visible (print string), Invisible (gap string).

3e1c Entities (for structure manipulation): Statement, Branch (statement plus all substructure), Group (sublist of branches), Plex (complete list of branches).

3e2 Structure may also be modified by joining statements, or breaking a statement into two at a specified point.

3e3 Generally, an operation and an entity make up a command, such as "Delete Word." To specify the command, the user types the first letter of each word in the command: thus "DW" specifies "Delete Word." There are occasional cases where a third word is used or where the first letter cannot be used because of ambiguities.

3f **File Output**

3f1 Files may be sent to any of a number of different output devices to produce hard copy—an upper/lower-case line printer, an

on-line high-quality typewriter, or paper tape to drive various typewriters.

3f1a In future it will be possible to send files via magnetic tape to an off-line CRT-to-film system from which we can produce Xerox prints, Multilith masters, or microform records.

3f2 Flexible format control may be exercised in this process by means of specially coded directives embedded in the files—running headers, page numbering, line lengths, line centering, suppression of location numbers, indenting, right justification (hyphenless), etc., are controllable features.

3g Compiling and Debugging

3g1 Source-code files written in any of our compiler languages (see below), or in the SDS 940 assembly language (ARPAS, in which our compiler output is produced) may be compiled under on-line control. For debugging, we have made a trivial addition to the SDS 940's DDT loader-debugger so as to operate it from the CRT displays. Though it was designed to operate from a Teletype terminal, this system gains a great deal in speed and power by merely showing with a display the last 26 lines of what would have been on the Teletype output.

3h Calculating

3h1 The same small innovation as mentioned above for DDT enables us to use the CAL system from a display terminal.

3i Conferencing

3i1 We have set up a room specially equipped for on-line conferencing. Six displays are arranged in the center of a square table (see Figure 6) so that each of twenty participants has good visibility. One participant controls the system, and all displays show the same view. The other participants have mice that control a large arrow on the screen, for use as a pointer (with no control function).

3i2 As a quick means of finding and displaying (with appropriate forms of view) any desired material from a very large collection, this system is a powerful aid to presentation and review conferences.

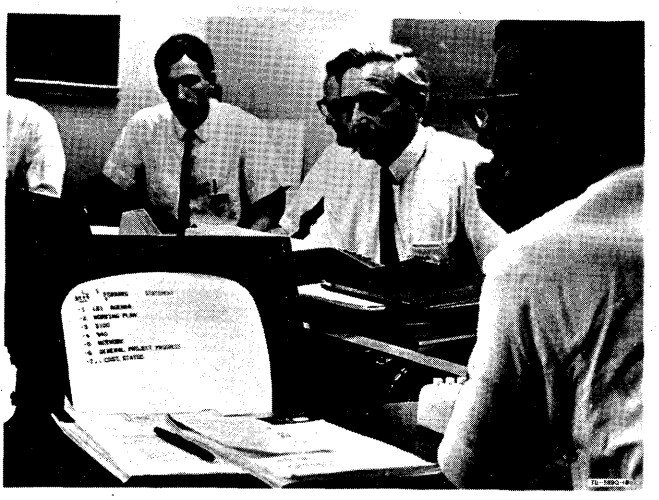


FIGURE 6—On-line conference arrangement

3i3 We are also experimenting with it in project meetings, using it not only to keep track of agenda items and changes but also to log progress notes, action notes, etc. The review aid is of course highly useful here also.

3i4 We are anxious to see what special conventions and procedures will evolve to allow us to harness a number of independent consoles within a conference group. This obviously has considerable potential.

4 SERVICE-SYSTEM SOFTWARE

4a The User's Control Language

4a1 Consider the service a user gets from the computer to be in the form of discrete operations—i. e., the execution of individual "service functions" from a repertoire comprising a "service system."

4a1a Examples of service functions are deleting a word, replacing a character, hopping to a name, etc.

4a2 Associated with each function of this repertoire is a "control-dialogue procedure." This procedure involves selecting a service function from the repertoire, setting up the necessary parameter designations for a particular application, recovering from user errors, and calling for the execution of the function.

4a2a The procedure is made up of the sequence of keystrokes, select actions, etc.

made by the user, together with the interspersed feedback messages from the computer.

4a3 The repertoire of service functions, together with their control-dialogue procedures, constitutes the user's "control language." This is a language for a "master-slave" dialogue, enabling the user to control application of the computer's capabilities to his own service.

4a3a It seems clear that significant augmentation of one's intellectual effectiveness from the harnessing of computer services will require development of a broad and sophisticated control-language vocabulary.

4a3b It follows that the evolution of such a control language is a very important part of augmentation-system research.

4a4 For the designer of user systems, it is important to have good means for specifying the nature of the functions and their respective control-dialogue procedures, so that a design specification will be

4a4a Concise, so that its essential features are easily seen

4a4b Unambiguous, so that questions about the design may be answered clearly

4a4c Canonical, so that information is easily located

4a4d Natural, so that the form of the description fits the conceptual frame of the design

4a4e Easy to compose, study, and modify, so that the process of evolutionary design can be facilitated.

4a5 It is also important for the user to have a description of the service functions and their control-dialogue procedures.

4a5a The description must again be concise, unambiguous, canonical, and natural; furthermore, it must be accurate, in that everything relevant to the user about the service functions and their control-dialogue procedures is described, and everything described actually works as indicated.

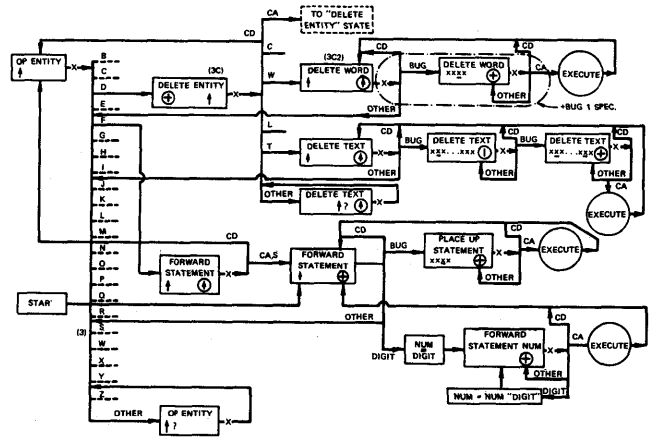


FIGURE 7—State-chart portrayal of part of the text-manipulation control language

4b State-Chart Representation of Control-Language Design

4b1 Figure 7 shows a charting method that was used in earlier stages of our work for designing and specifying the control-procedure portions of the control language. Even though limited to describing only the control-dialogue procedures, this representation nonetheless served very well and led us to develop the successive techniques described below.

4b2 Figure 7 shows actual control procedures for four service functions from the repertoire of an interactive system: Delete Word, Delete Text, Place Up Statement, and Forward Statement.

4b2a The boxes contain abbreviated descriptions of relevant display-feedback conditions, representing the intermediate states between successive user actions. Both to illustrate how the charting conventions are used and to give some feeling for the dynamics of our user-system control procedures, we describe briefly below both the chart symbols and the associated display-feedback conventions that we have developed.

4b2a1 The writing at the top of each box indicates what is to be shown as "command feedback" at the top of the display (see Figures 3, 4 and 5).

4b2a1a An uparrow sometimes ap-

pears under the first character of one of the words of Command Feedback.

4b2a1a1 This indicates to the user that the next character he types will be interpreted as designating a new term to replace that being pointed to—no uparrow under Command Feedback signifies that keyboard action will not affect the command designation.

4b2a1b "Entity" represents the entity word (i.e., "character," "word, "statement," etc.) that was last used as part of a fully specified command.

4b2a1b1 The computer often "offers" the user an entity option.

4b2a2 The circle in the box indicates the character to be used for the "bug" (the tracking spot), which alternates between the characters uparrow and plus.

4b2a2a The uparrow indicates that a select action is appropriate, and the plus indicates that a select action is inappropriate.

4b2a3 The string of X's, with underlines, indicates that the selected characters are to be underlined as a means of showing the user what the computer thinks he has selected.

4b2b There is frequently an X on the output line from a box on the chart. This indicates that the computer is to wait until the user has made another action.

4b2b1 After this next action, the computer follows a branching path, depending upon what the action was (as indicated on the chart) to reach another state-description box or one of the function-execution processes.

4c The Control Metalanguage

4c1 In search for an improvement over the state chart, we looked for the following special features, as well as the general features listed above:

4c1a A representational form using structural text so as to harness the power of our on-line text-manipulation techniques for composing, studying, and modifying our designs.

4c1b A form that would allow us to specify the service functions as well as the control-dialogue procedures.

4c1c A form such that a design-description file could be translated by a computer program into the actual implementation of the control language.

4c2 Using our Tree Meta compiler-compiler (described below), we have developed a next step forward in our means of designing, specifying, implementing and documenting our on-line control languages. The result is called "Control Metalanguage" (CML).

4c2a Figure 8 shows a portion of the description for the current control language, written in Control Metalanguage.

4c2a1 This language is the means for describing both the service functions and their control-dialogue procedures.

4c2b The Control Metalanguage Translator (CMLT) can process a file containing such a description, to produce a corresponding version of an interactive system which responds to user actions exactly as described in the file.

4c3 There is a strong correspondence between the conventions for representing the control procedures in Control Metalanguage and in the state chart, as a comparison of Figures 8 and 7 will reveal.

4c3a The particular example printed out for Figure 8 was chosen because it specifies some of the same procedures as in Figure 7.

4c3b For instance, the steps of display-feedback states, leading to execution of the "Delete Word" function, can readily be followed in the state chart.

4c3b1 The steps are produced by the user typing "D," then "W," then selecting a character in a given word, and then hitting "command accept" (the CA key).

4c3b2 The corresponding steps are outlined below for the Control Metalanguage description of Figure 8, progressing from Statement 3, to Statement 3c, to

Statement 3c2, to Subroutine + BUG-SPEC, etc.

4c3b3 The points or regions in Figure 7 corresponding to these statements and subroutines are marked by (3), (3C), (3C2), and (+ BUG1SPEC), to help compare the two representations.

4c3c These same steps are indicated in Figure 8, starting from Statement 3:

4c3c1 "D" sets up the state described in Statement 3C

4c3c2 "W" sets up the state described in Statement 3C2

FIGURE 8—Metalanguage description of part of control language

```
3  (wc:) zap  case

   3A (b) [edit]  dsp(backward ↑es*) .  case
   •
   •
   •

   3B (c) [edit] dsp(copy ↑es*) :s true => <am>adj1: . case

      3B1   (c)  s*=cc dsp(↑copy character) e*=c,character +bug2spec
      +cdlim(b1,p1,p2,p3,p4) +cdlim(b2,p5,p6,p7,p8)
      +cpchtx(b1,p2,p4,p5,p6) ;

      3B2  (w)  s*=cw dsp(↑copy word) e*=w,word +bug2spec
      +wdr2(b1,p1,p2,p3,p4) +wdr2(b2,p5,p6,p7,p8)
      +cpwdvs(b1,p2,p4,p5,p6) ;

      3B3  (l)  s*=cl dsp(↑copy line) e*=l,line +bug2spec
      +ldlim(b1,p1,p2,p3,p4)  +ldlim(b2,p5,p6,p7,p8) :c st b1←sf(b1) p2,
      rif :p p2>p1 cr: then (cr) else (null) ,  p5 p6, p4 se(b1): goto
      [s]

      3B4  (v)  s*=cv dsp(↑copy visible) e*=v,visible +bug2spec
      +vdr2(b1,p1,p2,p3,p4) +vdr2(b2,p5,p6,p7,p8)
      +cpwdvs(b1,p2,p4,p5,p6) ;
      •
      •
      •
      3b10  endcase +caqm ;

   3C (d) [edit] dsp(delete ↑es*) . case

      3C1 (c)  s*=dc dsp(↑delete character) e*=c,character +bug1spec
      +cdlim(b1,p1,p2,p3,p4) +del;

      3C2 (w)   s*=dw dsp(↑delete word) e*=w,word +bug1spec +wdr
      (b1,p1,p2,p3,p4) +del ;

      3C3 (l)   s*=dl dsp(↑delete line) e*=l,line +bug1spec...
      •
      •
      •
```

4c3c3 The subroutine +BUG1SPEC waits for the select-word (1) and CA (2) actions leading to the execution of the delete-word function.

4c3c3a Then the TWDR subroutine takes the bug-position parameter and sets pointers P1 through P4 to delimit the word in the text data.

4c3c3b Finally, the + DEL subroutine deletes what the pointers delimit, and then returns to the last-defined state (i.e., to where S* = DW).

4d Basic Organization of the On-Line System (NLS)

4d1 Figure 9 shows the relationships among the major components of NLS.

4d2 The Tree Meta Translator is a processor specially designed to produce new translators.

4d2a There is a special language—the Tree Meta Language—for use in describing the translator to be produced.

4d2b A special Tree Meta library of subroutines must be used, along with the output of the Tree Meta Translator, to produce a functioning new translator. The same library serves for every translator it produces.

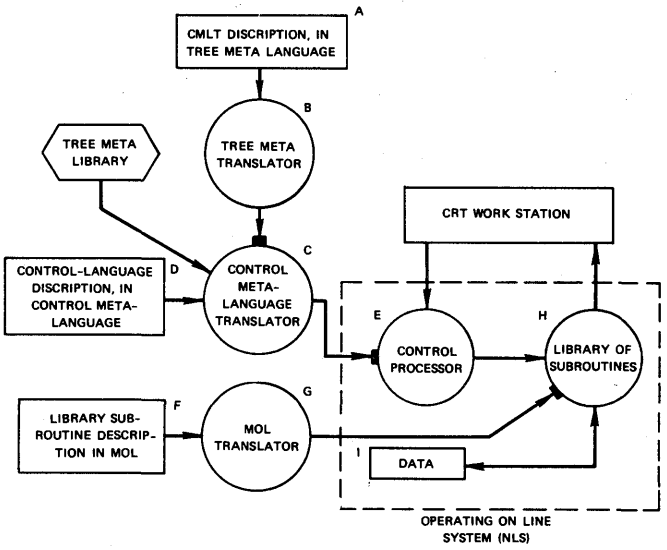


FIGURE 9—Basic organization of NLS showing use of compilers and compiler-compiler to implement it

4d3 For programming the various subroutines used in our 940 systems, we have developed a special Machine-Oriented Language (MOL), together with an MOL Translator to convert MOL program descriptions into machine code (see Ref. Hay1 for a complete description).

4d3a The MOL is designed to facilitate system programming, by providing a high-level language for iterative, conditional, and arithmetic operations, etc., along with a block structure and conventions for labeling that fit our structured-statement on-line manipulation aids.

4d3a1 These permit sophisticated computer aid where suitable, and also allow the programmer to switch to machine-level coding (with full access to variables, labels, etc.) where core space, speed, timing, core-mapping arrangements, etc., are critical.

4d4 The NLS is organized as follows (letters refer to Figure 9):

4d4a The Control Processor (E) receives and processes successive user actions, and calls upon subroutines in the library (H) to provide it such services as the following:

4d4a1 Putting display feedback on the screen

4d4a2 Locating certain data in the file

4d4a3 Manipulating certain working data

4d4a4 Constructing a display view of specified data according to given viewing parameters, etc.

4d4b The NLS library subroutines (H) are produced from MOL programs (F), as translated by the MOL Translator (G).

4d4c The Control Processor is produced from the control-language description (D), written in Control Metalanguage, as translated by the CMLT (C).

4d4d The CMLT, in turn, is produced from a description (A) written in Tree Meta, as translated by the Tree Meta Translator (B).

4d5 Advantages of Metalanguage Approach to NLS Implementation

4d5a The metalanguage approach gives us improved means for control-language specification, in terms of being unambiguous, concise, canonical, natural and easy to compose, study and modify.

4d5b Moreover, the Control Metalanguage specification promises to provide in itself a users' documentation that is completely accurate, and also has the above desirable characteristics to facilitate study and reference.

4d5c Modifying the control-dialogue procedures for existing functions, or making a reasonable range of changes or additions to these functions, can often be accomplished solely by additions or changes to the control-language record (in CML).

4d5c1 With our on-line studying, manipulating and compiling techniques, system additions or changes at this level can be thought out and implemented (and automatically documented) very quickly.

4d5d New functions that require basic operations not available through existing subroutines in the NLS library will need to have new subroutines specified and programmed (in MOL), and then will need new terms in CML to permit these new functions to be called upon. This latter requires a change in the record (A), and a new compilation of CMLT by means of the Tree Meta Translator.

4d5d1 On-line techniques for writing and modifying the MOL source code (F), for executing the compilations, and for debugging the routines, greatly reduce the effort involved in this process.

5 SERVICE-SYSTEM HARDWARE (OTHER THAN SDS 940)

5a In addition to the SDS 940, the facility includes peripheral equipment made by other manufacturers and equipment designed and constructed at SRI.

5b All of the non-SDS equipment is interfaced through the special devices channel which connects to the second memory buss through the SDS memory interface connection (MIC).

5b1 This equipment, together with the RAD, is a significant load on the second memory buss. Not including the proposed "special operations" equipment, the maximum expected data rate is approximately 264,000 words per second or one out of every 2.1 memory cycles. However, with the 940 variable priority scheme for memory access (see Pirtle[1]), we expect less than 1 percent degradation in CPU efficiency due to this load.

5b2 This channel and the controllers (with the exception of the disc controller) were designed and constructed at SRI.

5b2a In the design of the hardware serving the work stations, we have attempted to minimize the CPU burden by making the system as automatic as possible in its access to memory and by formatting the data in memory so as to minimize the executive time necessary to process it for the users.

5c Figure 10 is a block diagram of the special-devices channel and associated equipment. The major components are as follows:

5c1 Executive Control

5c1a This is essentially a sophisticated multiplexer that allows independent, asynchronous access to core from any of the 6 controllers connected to it. Its functions are the following:

5c1a1 Decoding instructions from the computer and passing them along as signals to the controllers.

5c1a2 Accepting addresses and requests for memory access (input or output) from the controllers, determining relative priority among the controllers, synchronizing to the computer clock, and passing the requests along to memory via the MIC.

5c1b The executive control includes a comprehensive debugging panel that allows any of the 6 controllers to be operated off-line without interfering with the operation of other controllers.

5c2 Disc File

5c2a This is a Model 4061 Bryant disc, selected for compatibility with the continued 940-system development by Berkeley's Project GENIE, where extensive file-handling software was developed.

5c2b As formatted for our use, the disc will have a storage capacity of approximately 32 million words, with a data-transfer rate of roughly 40,000 words per second and average access time of 85 milliseconds.

5c2c The disc controller was designed by Bryant in close cooperation with SRI and Project GENIE.

5c3 Display System

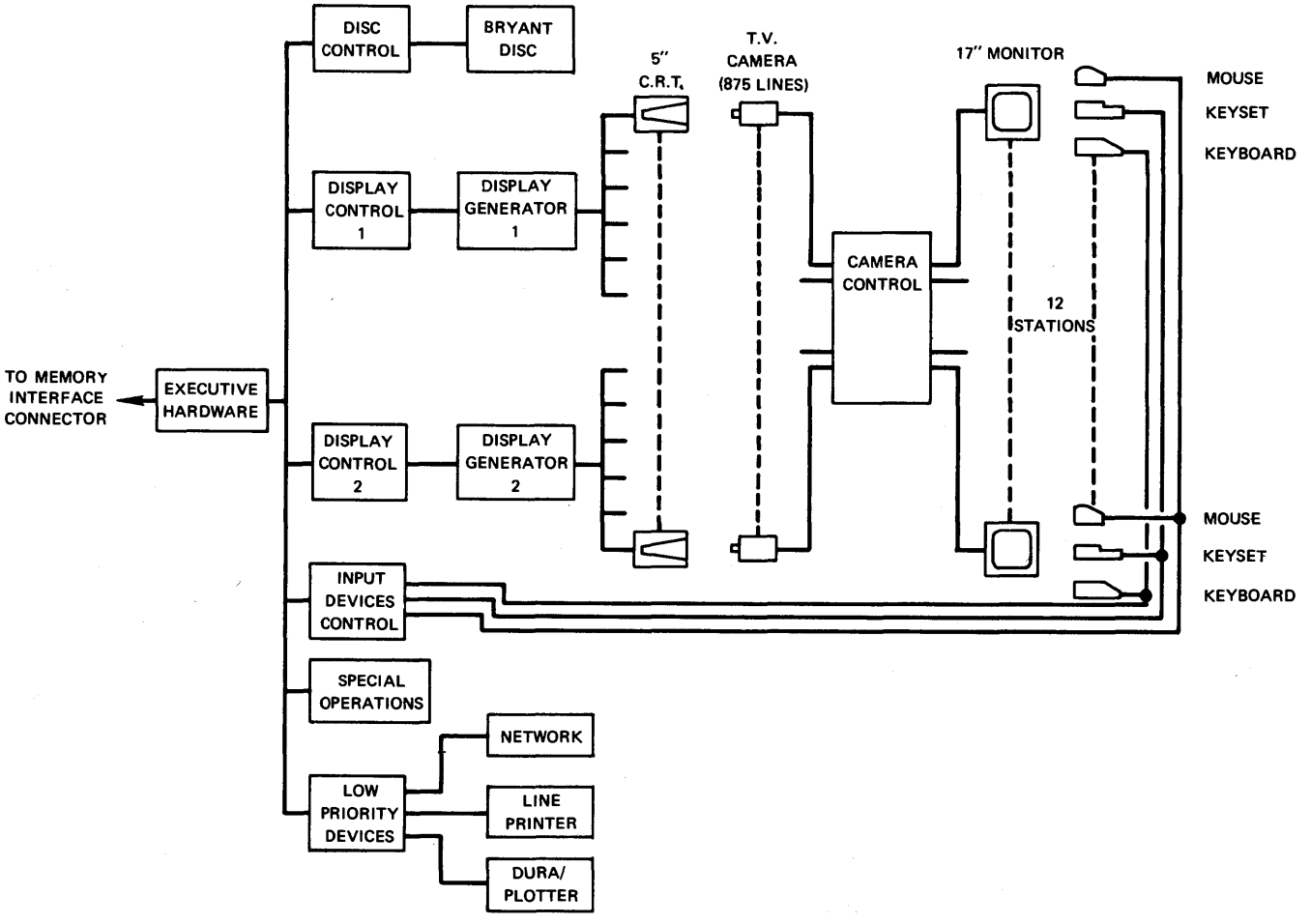
5c3a The display systems consists of two identical subsystems, each with display controller, display generator, 6 CRT's, and 6 closed-circuit television systems.

5c3b The display controllers process display-command tables and display lists that are resident in core, and pass along display-buffer contents to the display generators.

5c3c The display generators and CRT's were developed by Tasker Industries to our specifications. Each has general character-vector plotting capability. They will accept display buffers consisting of instructions (beam motion, character writing, etc.) from the controller. Each will drive six 5-inch high-resolution CRT's on which the display pictures are produced.

5c3c1 Character writing time is approxi-mately 8 microseconds, allowing an aver-

FIGURE 10—Special devices channel



age of 1000 characters on each of the six monitors when regenerating at 20 cps.

5c3d A high-resolution (875-line) closed-circuit television system transmits display pictures from each CRT to a television monitor at the corresponding work-station console.

5c3e This system was developed as a "best solution" to our experimental-laboratory needs, but it turned out to have properties which seem valuable for more widespread use:

5c3e1 Since only all-black or all-white signal levels are being treated, the scan-beam current on the cameras can be reduced to achieve a short-term image-storage effect that yields flicker-free TV output even when the display refresh rate is as low as 15 cps. This allows a display generator to sustain about four times more displayed material than if the users were viewing direct-view refreshed tubes.

5c3e2 The total cost of small CRT, TV camera, amplifier-controller, and monitor came to about $5500 per work station—where a random-deflection, display-quality CRT of similar size would cost considerably more and would be harder to drive remotely.

5c3e3 Another cost feature which is very important in some system environments favors this TV approach: The expensive part is centrally located; each outlying monitor costs only about $600, so terminals can be set up even where usage will be low, with some video switching in the central establishment to take one terminal down and put another up.

5c3e4 An interesting feature of the video system is that with the flick of a switch the video signal can be inverted, so that the image picked up as bright lines on dim background may be viewed as black lines on a light background. There is a definite user preference for this inverted form of display.

5c3f In addition to the advantages noted above, the television display also invites the use of such commercially available devices as extra cameras, scan converters, video switches, and video mixers to enrich system service.

5c3f1 For example, the video image of a user's computer-generated display could be mixed with the image from a camera focused on a collaborator at another terminal; the two users could communicate through both the computer and a voice intercom. Each user would then see the other's face superimposed on the display of data under discussion.

5c3f2 Superimposed views from cameras focused on film images or drawings, or on the computer hardware, might also be useful.

5c3f3 We have experimented with these techniques (see Figure 11) and found them to be very effective. They promise to add a great deal to the value of remote display terminals.

5c4 Input-Device Controller

5c4a In addition to the television monitor, each work-station console has a keyboard, binary keyset, and mouse.

5c4b The controller reads the state of these

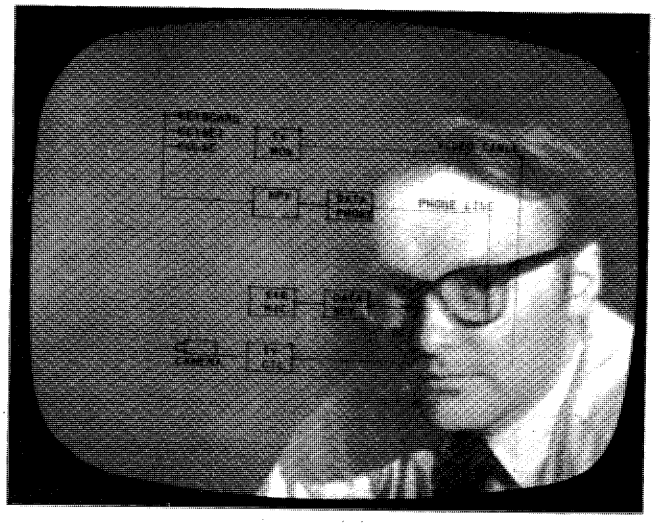


FIGURE 11—Television display obtained by mixing the video signal from a remote camera with that from the computer-generated display

devices at a preset interval (about 30 milliseconds) and writes it into a fixed location table in core.

5c4b1 Bits are added to information from the keyboards, keysets and mouse switches to indicate when a new character has been received or a switch has changed state since the last sample. If there is a new character or switch change, an interrupt is issued after the sample period.

5c4b2 The mouse coordinates are formatted as a beam-positioning instruction to the display generator. Provisions are made in the display controller for including an entry in the mouse-position table as a display buffer. This allows the mouse position to be continuously displayed without any attention from the CPU.

5c5 Special Operations

5c5a The box with this label in Figure 10 is at this time only a provision in the executive control for the addition of a high-speed device. We have tentative plans for adding special hardware here to provide operations not available in the 940 instruction set, such as character-string moves and string-pattern matching.

5c6 Low-Priority Devices

5c6a This controller accommodates three devices with relatively low data-transfer rates. At this time only the line printer is implemented, with provisions for adding an on-line typewriter (Dura), a plotter, and a terminal for the proposed ARPA computer network.

5c6a1 The line printer is a Potter Model HSP-3502 chain printer with 96 printing characters and a speed of about 230 lines per minute.

6 REFERENCES

6a (English 1) W K ENGLISH D C ENGELBART B HUDDART
Computer-aided display control
Final Report Contract NAS 1-3988 SRI Project 5061 Stanford Research Institute Menlo Park California July 1965

6b (English2) W K ENGLISH D C ENGELBART M L BERMAN
Display-selection techniques for text manipulation
IEEE Trans on Human Factors in Electronics Vol HFE-8 No 1 1967

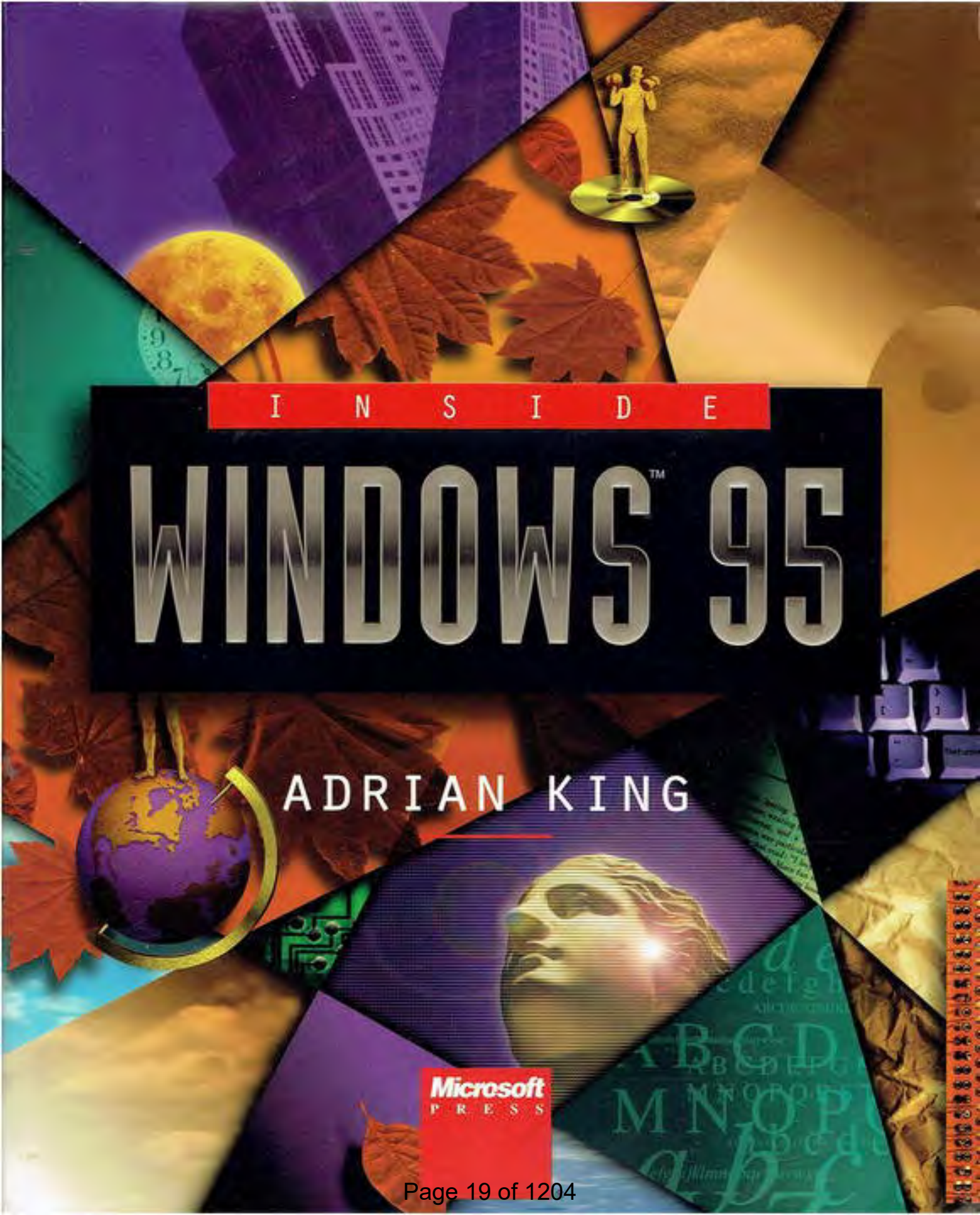
6c (Engelbart1) D C ENGLEBART
Augmenting human intellect: A conceptual framework
Summary Report Contract AF 49 638 1024 SRI Project 3578 Stanford Research Institute Menlo Park California October 1962

6d (Engelbart2) D C ENGELBART
A conceptual framework for the augmentation of man's intellect
In Vistas in Information Handling Vol 1 D W Howerton and D C Weeks eds Spartan Books Washington D C 1663

6e (Hay1) R E HAY J F RULIFSON
MOL940 Preliminary speifications for an ALGOL *like machine-oriented language for the SDS 940*
Interim Technical Report Contract NAS 1-5940 SRI Project 5890 Stanford Research Institute Menlo Park California March 1968

6f (Pirtle1) M PIRTLE
Intercommunication of Processors and memory
Proc Fall Joint Computer Conference Anaheim California November 1967

APPENDIX B


INSIDE
WINDOWS™ 95
ADRIAN KING
Microsoft
PRESS

I N S I D E

WINDOWS™ 95

ADRIAN KING

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 1994 by Adrian King

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
King, Adrian, 1953-
Inside Windows 95 / Adrian King.
p. cm.
Includes index.
ISBN 1-55615-626-X
1. Windows (Computer programs) 2. Microsoft Windows (Computer file) I. Title.
QA76.76.W56K56 1994
005.4'469--dc20 93-48485
CIP

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QMQM 9 8 7 6 5 4

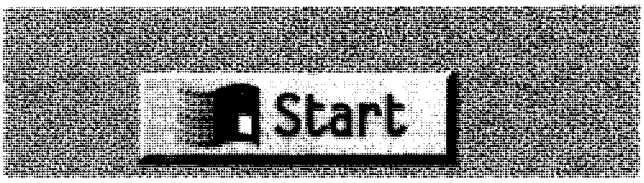
Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (206) 936-7329.

PageMaker is a registered trademark of Aldus Corporation. Apple, AppleTalk, LaserWriter, Mac, Macintosh, and TrueType are registered trademarks of Apple Computer, Inc. LANtastic is a registered trademark of Artisoft, Inc. Banyan and Vines are registered trademarks of Banyan Systems, Inc. Compaq is a registered trademark of Compaq Computer Corporation. CompuServe is a registered trademark of CompuServe, Inc. Alpha AXP, DEC, and Pathworks are trademarks of Digital Equipment Corporation. LANstep is a trademark of Hayes Microcomputer Products, Inc. HP and LaserJet are registered trademarks of Hewlett-Packard Company. Intel is a registered trademark and EtherExpress, Pentium, and SX are trademarks of Intel Corporation. COMDEX is a registered trademark of Interface Group-Nevada, Inc. AS/400, IBM, Micro Channel, OS/2, and PS/2 are registered trademarks and PC/XT is a trademark of International Business Machines Corporation. 1-2-3, Lotus, and Notes are registered trademarks of Lotus Development Corporation. Microsoft, MS, MS-DOS, and XENIX are registered trademarks and ODBC, Win32s, Windows, Windows NT, and the Windows operating system logo are trademarks of Microsoft Corporation. MIPS is a registered trademark and R4000 is a trademark of MIPS Computer Systems, Inc. NetWare and Novell are registered trademarks of Novell, Inc. Soft-Ice/W is a registered trademark of Nu-Mega Technologies, Inc. DESQview is a registered trademark and Qemm is a trademark of Quarterdeck Office Systems. OpenGL is a trademark of Silicon Graphics, Inc. PC-NFS, Sun, and Sun Microsystems are registered trademarks of Sun Microsystems, Inc. TOPS is a registered trademark of TOPS, a Sun Microsystems company. UNIX is a registered trademark of UNIX Systems Laboratories.

Acquisitions Editor: Mike Halvorson
Project Editor: Erin O'Connor
Technical Editors: Seth McEvoy and Dail Magee, Jr.



CONTENTS SUMMARY

Foreword . . . . . *xvii*
Preface . . . . . *xxi*
Introduction . . . . . *xxv*

CHAPTER ONE
THE ROAD TO CHICAGO **1**
CHAPTER TWO
INTEL PROCESSOR ARCHITECTURE **33**
CHAPTER THREE
A TOUR OF CHICAGO **63**
CHAPTER FOUR
THE BASE SYSTEM **103**
CHAPTER FIVE
THE USER INTERFACE AND THE SHELL **157**
CHAPTER SIX
APPLICATIONS AND DEVICES **223**
CHAPTER SEVEN
THE FILESYSTEM **275**
CHAPTER EIGHT
PLUG AND PLAY **309**
CHAPTER NINE
NETWORKING **341**
CHAPTER TEN
MOBILE COMPUTING **381**
EPILOGUE
LEAVING CHICAGO **407**
Glossary *427*
Index *455*

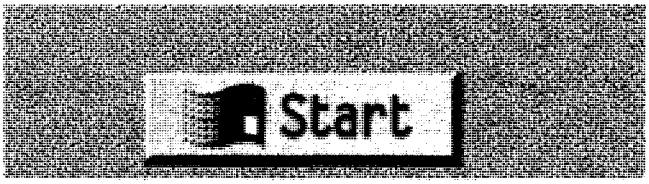


TABLE OF CONTENTS

Foreword . . . . . *xvii*
Preface . . . . . *xxi*
Introduction . . . . . *xxv*

CHAPTER ONE

THE ROAD TO CHICAGO **1**

The Mission for Windows 95 3
Help for the End User 3
Hardware Platforms 4
For the Developer—32 Bits at Last 5
Shall We Go to Chicago or Cairo? 6
First Stop—Chicago 7
Clients and Servers 8
And On to Cairo 10
Summary 12
Project Goals 13
Compatibility 14
The Compatibility Fallback 15
Performance 16
Robustness—Adieu UAE? 17
Timely Product Availability 18
Easy Setup and Configuration 19
The Plug and Play Initiative 20
Configuring Windows 21
User-Level Operations 21
New Shell and User Interface 22
The New Shell 22
Complete Protected Mode Operating System 23
32-Bit Application Support 24
The Jump to 32 Bits 26
Networking and Mobile Computing 27

Bringing Windows 95 to Market 28
For Microsoft—The Bottom Line 30
Conclusion 31

CHAPTER TWO

INTEL PROCESSOR ARCHITECTURE **33**

Intel Inside 34
The Intel Processor Family 35
Backward Compatibility 36
Processor Architecture 37
The 8080 and 8086 Processors 38
The 640K Barrier 39
The 80286 Processor 41
The 80386 Processor 43
80386 Memory Addressing 45
80386 Descriptor Format 45
The Descriptor in Summary 48
Virtual Memory 48
Virtual Memory Management 49
Good Virtual Memory Management 50
Mixing 286 and 386 Programs 54
The Protection System 54
Memory Protection 55
Operating System Protection 56
Device Protection 57
Low-Level Device Access 57
High-Level Device Access 58
Using the 80386 Device Protection Capabilities 59
Virtual 8086 Mode 60
Conclusion 61

CHAPTER THREE

A TOUR OF CHICAGO **63**

System Overview 63
The Base System 66
Windows and Modes 67

Virtual Machines 68
Windows Virtual Machines 70
Initialization 70
The System Virtual Machine 71
MS-DOS Virtual Machines 72
Protected Mode MS-DOS Applications 73
DPMI 74
Multitasking and Scheduling 75
Multitasking Models 76
Critical Sections 79
Processes in Windows 80
Modules 80
API Support 81
Dynamic Linking 82
Support from the Base System 84
Memory Management 85
Application Virtual Memory 86
Heap Allocation 87
Windows 95 Application Memory Management 87
System Memory Management 88
Windows Device Support 90
Device Virtualization 90
Minidrivers 91
The Windows Interface 92
What Is a Window? 92
Windows 95 User Interface Design 95
Windows Programming Basics 96
Event Driven Programming 96
Message Handling 97
Program Resources 99
Windows 95 Programming 99
Conclusion 101
References 101

CHAPTER FOUR

THE BASE SYSTEM **103**
Windows 95 Diagrammed 104

Windows 95 Surveyed 106
Protection Rings in Windows 95 107
Windows 95 Memory Map 108
Tasks and Processes 110
Virtual Machine Management 111
Real MS-DOS 111
Virtual Machine Scheduling 112
The Windows 95 Schedulers 114
Scheduling Within the System Virtual Machine 116
Controlling the Scheduler 116
Threads and UAEs 117
Threads and Idle Time 118
Application Message Queues 119
Physical Memory Management 121
Virtual Memory Management 125
Memory Mapped Files 127
Reserving Virtual Address Space 128
Private Heaps 129
Virtual Machine Manager Services 129
Calling Virtual Machine Manager Services 131
VMM Callbacks 131
Loading VxDs 132
The Shell VxD 134
Getting Around in Ring Zero 135
Calling Windows 95 Base OS Services 137
Calling from One VxD to Another 138
VMM Service Groups 140
Application Support 141
The API Layer 142
Mixing 16-Bit and 32-Bit Code 143
The Win32 Subsystem 147
Internal Synchronization 149
Conclusion 155
References 156

CHAPTER FIVE

THE USER INTERFACE AND THE SHELL **157**

Improving on Windows 3.0 and 3.1 159
System Configuration and Control 160
Program Manager, File Manager, Task Manager 160
Control Functions 162
Consistency 162
Visuals 164
Scalability 164
Concepts Guiding the New User Interface 165
The Document-Centric Interface 166
Look and Feel 167
The Windows 95 Shell 169
Folders and Shortcuts in the Windows 95 Shell 170
Desktop Folders 172
System Setup 173
The Initial Desktop 174
The Desktop 177
The Taskbar 179
On-Screen Appearance 182
Light Source 184
Property Sheets 185
Online Help 186
Implementation 188
Design Retrospective 189
The Outside Influences 189
The Development of the Shell 190
Changes in the Shell 192
The Taskbar 194
Folders and Browsing 195
Animation 196
The Transfer Model 197
Other Changes 198
The New Appearance 198
Screen Appearance 198
Visual Elements 201
Scalability 201

Menus 202
Window Buttons 204
Icons 204
Proportional Scroll Box and Sizing Handle 205
New Controls 205
Tool Bar Control 205
Button List Box Control 206
Status Window Control 206
Column Heading Control 207
Progress Indicator Control 208
Slider Control 208
Spin Box Control 208
Rich Text Control 209
Tab Control 209
Property Sheet Control 209
List View and Tree View Controls 210
New Dialog Boxes 210
File Open Dialog 211
Page Setup Dialog 213
Long Filenames 213
Windows 95 Support for MS-DOS Applications 215
Application Guidelines for Windows 95 217
Follow the Style Guidelines 218
Support Long Filenames 218
Support UNC Pathnames 218
Register Document and Data Types, and Support Drag and Drop 218
Use Common Dialogs 219
Reduce Multiple Instances of an Application 219
Be Consistent with the Shell 219
Revise Online Help 219
Support OLE Functionality 220
Conclusion 220
Reference 221

CHAPTER SIX

APPLICATIONS AND DEVICES **223**
The Win32 API 224
Goals for Win32 226
Components of the Win32 API 227
The Win32 API on Windows 95 229
Porting to the Win32 API 229
Porting Tools 229
API Changes 230
Memory Management 232
Version Checking 233
Nonportable APIs 233
Win32 on Windows 95 234
Security APIs 234
Console APIs 235
32-Bit Coordinate System 235
Unicode APIs 235
Server APIs 236
Printer Support 236
Service Control Manager APIs 236
Event Logging 236
Detailed Differences 237
Programming for Windows 95 238
Multitasking 238
Memory Management 241
Plug and Play Support 241
The Registry 242
The User Interface 245
OLE 245
International Support 248
Structured Exception Handling 249
The Graphics Device Interface 252
GDI Architecture 255
Performance Improvements 256
Limit Expansion 256
New Graphics Features 257

TrueType 258
Metafile Support 258
Image Color Matching 259
Color Profiles 261
Communicating Color Information 261
The Display Subsystem 262
The DIB Engine 265
The Display Mini-Driver 266
Bank-Switched Video Adapters 267
Interfacing with the DIB Engine 268
The Printing Subsystem 269
Printing Architecture 270
The Printing Process 270
Using the Universal Printer Driver 272
Conclusion 274
References 274

CHAPTER SEVEN

THE FILESYSTEM **275**

Overview of the Architecture 277
Long Filename Support 281
Storing Long Filenames 282
Generating Short Filenames 288
MS-DOS Support for Long Filenames 289
Long Filenames on Other Systems 291
Installable Filesystem Manager 291
Calling a Filesystem Driver 293
Filesystem Drivers 294
FSD Entry Points 295
I/O Subsystem 296
Device Driver Initialization 298
Controlling an I/O Request 299
Calldown Chains 300
Asynchronous Driver Events 301
Interfacing to the Hardware 302
Initialization 302
Execution 303
Interrupt 303

Other Layers in the Filesystem Hierarchy 303
Volume Tracking Drivers 304
Type Specific Drivers 305
SCSI Manager 306
Real Mode Drivers 307
Conclusion 308
References 308

CHAPTER EIGHT

PLUG AND PLAY **309**
Why Do We Need Another Standard? 310
History of the Plug and Play Project 312
Goals for Plug and Play 314
Easy Installation and Configuration of New Devices 315
Support for a New Hardware Standard 315
New ISA Board Standard 317
Seamless Dynamic Configuration Changes 318
Compatibility with the Installed Base and Old Peripherals 319
Operating System and Hardware Independence 320
Reduced Complexity and Increased Flexibility of Hardware . . . 320
The Components of Plug and Play 321
How the Subsystem Fits Together 325
After a System Configuration Change 328
Hardware Tree 328
Device Nodes 329
Device Identifiers 331
Hardware Information Databases 332
Plug and Play Events 333
Configuration Manager 333
Enumerators 334
Resource Arbitrators 335
Plug and Play BIOS 336
Plug and Play Device Drivers 337
Applications in a Plug and Play System 338
Conclusion 339
References 340

CHAPTER NINE

NETWORKING **341**
Windows Networking History 342
Networking Goals 346
Network Software Architecture 347
WOSA 348
Network Layers 351
Network Operations 353
The Multiple Provider Router 355
32-Bit Networking APIs 357
Network Resources 357
Connection APIs 358
Enumeration APIs 359
Error Reporting APIs 360
Local Device Name APIs 360
UNC APIs 360
Password Cache API 360
Authentication Dialog API 361
Interfacing to the Network Provider 361
The Network Provider 362
Network Provider Services 363
Device Redirection SPI 364
Shell SPI 365
Enumeration SPI 365
Authentication SPI 366
Network Transports 366
Network Device Drivers 368
Network Driver Compatibility 369
Network Configurations 370
The Network Server 372
Server Components 373
Network Printing 375
Network Security 377
Access Controls 378
Share-Level Security 379
User-Level Security 379
Conclusion 379
Reference 380

CHAPTER TEN

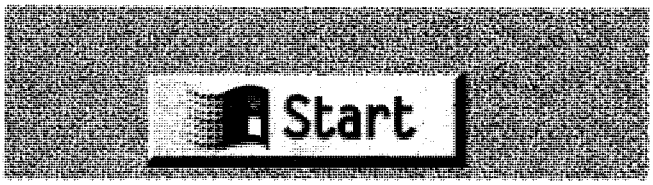
MOBILE COMPUTING **381**

Remote Communications Support 382
Remote Network Access 385
Types of Remote Access 386
The Telephony API 389
Telephony Applications 390
Modem Support 391
The Communications Driver 392
The Info Center 394
Info Center Applications 396
Messaging APIs 396
Messaging Service Providers 397
Portable System Support 398
Power Management 398
Docking Station Support 399
File Synchronization 400
The Briefcase API 403
Conclusion 404

EPILOGUE

LEAVING CHICAGO **407**

Glossary *427*
Index *455*



FOREWORD

I first met Adrian King in 1981, on the floor of a trade show in Amsterdam. I was new to Microsoft—a small company of 75 people with $7.5 million in revenues—and I was on my first trip to Europe to meet customers and distribution partners. The trade show turned out to be a flop—more exhibitors than customers. Adrian and I by ourselves might have outnumbered the customers.

We had a lot of time to talk to each other, and I found out that Adrian had graduated from the University of Liverpool with a master's degree in computer science and had joined Logica, a big European consulting outfit, straight out of school. It was clear right off that he understood technology and a lot else besides.

We tried to figure out why the aisles were so empty, and that got us into talking about the future for software. I remember thinking that Adrian was an impressive guy and reflecting that with more people like Adrian involved, the software business might really take off. But even in our freewheeling exchange of ideas, we didn't come close to envisioning today's incredible market for software.

A little later, Adrian managed to convince Logica to branch out from their consulting business into software products—no small feat at the time—and they became Microsoft's European XENIX partner. Through the early 1980s, Adrian and I worked together to develop the European XENIX business. Then, in April of 1984, we met to review XENIX support issues. That's how it started out, anyway. During the first half of the meeting, Adrian did his best to convince me that Microsoft had to do a number of different things to improve our XENIX product support. During the second half, I did my best to convince Adrian that he really ought to become our XENIX product manager and take care of those things himself. With a little help from Bill Gates, I was able to persuade Adrian to do just that.

Adrian did a great job, and before long we gave him even more to do. He eventually became our director of operating systems products, picking up responsibilities for MS-DOS and Microsoft OS/2 as well as

XENIX. At the same time I was focusing on Windows, which had become a big priority for the company. We had come to believe that using a mouse with a graphical user interface was a natural, intuitive way to use a computer. Adrian worked on the early Windows projects, and in November of 1985 I put him in charge of Windows/386.

The effort we put in on the early versions of Windows was a foundation for the blockbuster success of Windows 3.0 and Windows 3.1. The work that Adrian and the rest of the team did on the Windows/386 project formed the basis for much of Microsoft's MS-DOS support in Windows 3.1 and even in Windows NT, for example. And many of the people from that Windows/386 team are still involved in our Windows development today.

Adrian went on to other important projects at Microsoft, and then in 1991 he left to pursue his interest in peer-to-peer networking at a smaller company. I'm sure that if Adrian were still at Microsoft he'd be deeply involved in the development of Windows 95. But at least he's back in the Microsoft orbit—this time as a chronicler, the author of *Inside Windows 95*.

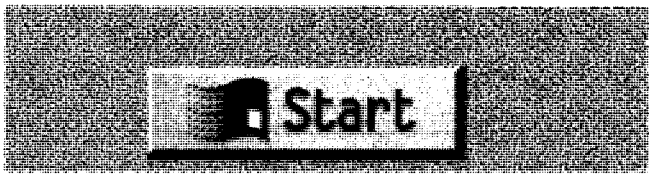
Microsoft's goals for Windows 95 are the same goals we've had for every release of Windows. We want to make computing even easier. We want to increase end user productivity. We want to provide a development platform for the desktop. We want to provide a high-volume, low-cost operating system that will spur industry growth and innovation. We believe that Windows 95 will accomplish these goals and that Windows 95 will be even more important to the PC world than Windows 3.1, which now has over 60 million users.

The list of great new features for Windows 95, a true 32-bit operating system, is amazingly long. Windows 95 will offer a vastly improved user interface, true multitasking, a freshly designed filesystem, better connectivity, better support for notebook PCs, easier installation and configuration—all with performance at least as good as Windows 3.1 performance.

I'm very excited that Adrian has written this book about our most important Windows operating system ever. We're lucky that Adrian turned out to be a good writer too because he has a perspective that only someone from the old days could bring to bear on the history and the accomplishments of the "Chicago" Windows project. Everyone will want to read *Inside Windows 95*—the interested power user, solution providers, developers, and administrators. I heartily recommend this

book to anyone who will want to take full advantage of the technological innovations in Windows 95. Adrian does an excellent job of explaining the major architectural components of the system and provides a lot of insight into the thinking behind the design and implementation of Windows 95. I've greatly enjoyed reading his account of the project and the product in this book, and I think you will too.

Steve Ballmer
Executive Vice President, Microsoft
Redmond, Washington
August 1994



PREFACE

Writing a book about a yet to be released software product and publishing it before the product even ships has to be asking for trouble. Throw in other factors such as the fact that the product in question is one that literally thousands of people will examine and critique in minute detail, and you can easily build a case for declining the writing opportunity. So, of course, I accepted. *Inside Windows 95* is the result.

When I started working for Microsoft in 1984, I'd already known the company as a customer and development partner for a few years. One thing I'd learned very quickly about Bill Gates and Steve Ballmer is that they never, ever give up on something they believe in. In 1984 and 1985, even with massive delays in its initial planned shipment, Windows was the something they weren't giving up on. My first office at Microsoft was next to Steve Ballmer's. One day, after more bad news about Windows shipment dates, he and his assistant packed everything up and moved downstairs to occupy new offices in the midst of the Windows development team (a group maybe ten strong at the time). Steve was now the Windows project manager, and he wasn't about to give up.

Windows 1.0 eventually shipped in late 1985. Describing the market's reaction as lukewarm is akin to describing Bill Gates as well off. I remember installing the first Windows Software Development Kit on an IBM PC XT and being at different moments impressed by its features and bewildered by its complexity. Looking back on it now, I can see that it was of course sheer madness for Microsoft to believe that Windows could succeed on the limited hardware available at the time.

But Microsoft wasn't about to give up. Through successive versions, Windows gradually got better and the hardware got faster and more capacious. In 1987 and 1988 I managed the project that produced Windows/386 and launched it on the first 386-based PC: the Compaq Deskpro. It was my favorite time at Microsoft, and the entire project team—all fifteen of us—were rather proud of Windows/386. In comparison to MS-DOS it still didn't sell worth a darn. Even Steve Ballmer was beginning to think that OS/2 might be the right strategy.

But Microsoft didn't give up, and on May 22, 1990, Bill Gates introduced the latest and greatest release of Windows—version 3.0—to a rapt audience in New York City. Things were different this time. It was obvious to me in the theater that day that Windows was about to become a seven-year-old overnight success. And it did. Bill and Steve would probably try to convince you it was planned that way. Don't believe it. Whether the galaxies were finally in correct alignment, or a confluence of market factors finally came about, or sheer determination finally carried the day is no longer relevant—Windows was finally a hit.

I was involved only a little in the development of Windows 3.0 and not at all in the development of Windows 3.1. Shortly before I left Microsoft in 1991, I began working on what was eventually to become part of the base operating system for Windows 95. Clearly I was not destined to escape the project entirely, and the opportunity to write this book on Windows 95 for Microsoft Press is one I've enjoyed a lot. Watching a Windows release once again is fascinating. The scope of the work that goes into a major new release of Windows these days is staggering, with hundreds of people involved rather than only a few dozen.

Of course, I'm only writing about what many have built and others have yet to go out and sell. Although the Windows team at Microsoft is considerably bigger these days, it still includes a few people from back when Steve Ballmer was the project manager. And Steve's current role at Microsoft as Executive Vice President of Sales and Support means that he is now in charge of the worldwide sales campaign for Windows 95. Windows 95 will enter the market under some competitive pressure. Proponents of UNIX, OS/2, and NetWare certainly haven't relaxed their attempts to improve their own products and their market shares. But Windows 95 is definitely the product to beat. I'm quite sure Steve won't give up on this challenge either—which means that nothing has really changed since 1985 except the location of Steve's office and the size of his marketing budget.

Special thanks go to Erin O'Connor and her team at Microsoft Press for overcoming my English and several other obstacles in the preparation of this book. Claudette Moore and Mike Halvorson got the project started, and several people at Microsoft gave time and assistance to the project, for which I'm grateful. George Moore and Joe Belfiore in particular were always willing to answer my questions. It has been more than a year since I began work on this book, and, as I write, I know there's still a lot of work left to finish Chicago. That effort is but a tiny part of the total still needed to ship Chicago and make it a suc-

cess. The industry magazines have already published their first reviews of the Chicago Beta-1 release. IBM has launched its anti-Chicago advertising campaign. The pundits and self-styled experts have begun their critique of a product that won't be in the stores for months yet. Windows 95 has a long way to go before it will be a runaway success. But I'm sure that will happen. Microsoft won't give up before it does.

If you'd like to talk to me about this book or about Windows 95 in general, I'm readily available on the Internet as *adriank@gravity.wa.com.* I hope you find at least some of the book useful and enjoyable. Thanks for taking the time to read it.

Adrian King
July 12, 1994

Publisher's Note

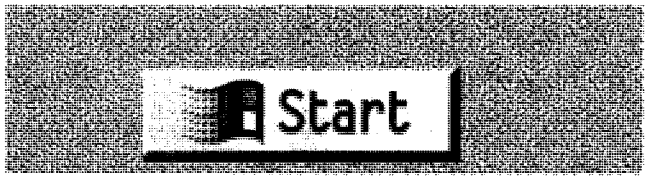
As we went to press, some aspects of Windows 95 were still under a general nondisclosure agreement, but Microsoft had made public a great deal of information about Windows 95. This book offers an interpretation of that information, and the author's conclusions are based on his exploration of Beta-1. The "Chicago" story continues to unfold, and the product will continue to be refined. For up-to-the-minute changes in information on Windows 95, we recommend that you periodically visit the WIN_NEWS forum, which you can find at the following locations:

On CompuServe:	*GO WINNEWS*
On the Internet:	*ftp://ftp.microsoft.com/PerOpSys/Win_News/Chicago*
	http://www.microsoft.com
On AOL:	keyword *WINNEWS*
On Prodigy:	jumpword *WINNEWS*
On Genie:	*WINNEWS* file area on Windows RTC

You can also subscribe to Microsoft's electronic newsletter *WinNews*. To subscribe, send Internet e-mail to *enews@microsoft.nwnet.com* and put the words *SUBSCRIBE WINNEWS* in the text of the e-mail.

When Windows 95 is released, be sure to head to your bookstore for complete accounts of developing for and using Windows 95.

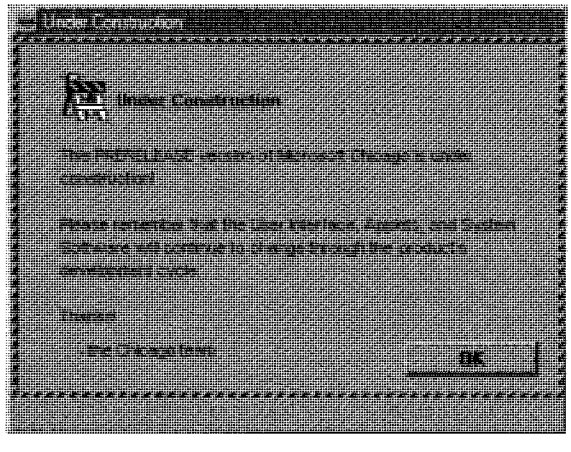
Microsoft Press
September 16, 1994



INTRODUCTION

To describe this book as an account of everything you could possibly want to know about Windows 95, or indeed as an account of everything in Windows 95, would be to mislead. The sheer scope of the Windows 95 development project makes it impossible to write the only book about the product you'll ever need to buy. If you're an avid student of Windows, I'm sure your sagging bookshelf will have to bear further strain in the months ahead. If you're a regular user, you'll find a whole host of new and exciting features to explore in Windows 95.

First a warning. Even as I write, Windows 95 is still in development and scheduled for release a few months into the future. Microsoft made the first external release of the product in August 1993. After installation, one of the first icons you were tempted to double-click on produced this unsettling screen:



In many other places in the product you could find similar warnings: *subject to change, not yet implemented,* and so on. It seems appropriate to use the Under Construction screen at the front of this book. My warnings won't be as dire, though, since this book does describe features you really can expect to find when Windows 95 hits the streets late this year. This book is current as of the Chicago Beta-1 release that

Microsoft shipped in June 1994. By and large the product was feature complete at the time of that beta release. However (and here's that warning), since the book has to go to the printer before the product ships, there will undoubtedly be some changes of detail in the final release of the product. And the incompatible goals of exploring every last feature of Windows 95 and still producing this book in advance of the product means that some features won't be examined in much detail and some features will be left out altogether.[1]

The intention of the book is to provide a technical introduction to the Windows 95 system, including enough detail to satisfy any Windows user and most system administrators and Windows programmers. The book is also "Inside Windows 95," meaning that the emphasis is on what the system can do, how it does it, and why its features were designed and implemented in particular ways. If you're looking for a book that teaches you how to use the Windows 95 interface, how to customize Windows 95, or how to write Windows 95 applications, this book isn't it. But this book does give you a thorough analysis of the system architecture and explores every important new feature of Windows 95.

Windows 95 is a major product release for Microsoft. It incorporates significant new features for exploitation by developers, and major advances in the user interface and in system usability that should benefit the end user. Since Microsoft Windows has become such an immensely successful product, new releases bear a burden of backward compatibility. Windows 95 has to carry forward the MS-DOS legacy. And Windows 95 isn't Microsoft's only Windows family operating system. Windows 95 must take its place alongside Windows NT and the forthcoming Cairo system. Chapter One explores the goals of the Windows 95 project, the constraints on the development team, the market for the product, and the role of Windows 95 in Microsoft's overall systems software strategy.

When I began work on this book, Microsoft's internal planning had Windows 95 shipping at the end of the year—the year 1993. Windows 95 would have been truly unique among operating systems if it had shipped on the originally planned date. As I write, the testing status of Windows 95 suggests that there's a reasonable chance that it will indeed ship at the end of the year—the year 1994.

1. One major "change" that did make it into this book is the Windows 95 name. Everyone had been assuming that Chicago's real name would be Windows 4.0. In July 1994 Microsoft decided on the Windows 95 name to align the operating system with a planned company-wide revision of product names. Fortunately, they made the decision just before the book went to press.

One aspect of the product that I can't cover in this book is exactly how Windows 95 will be packaged and priced. Microsoft executives are characteristically vague about these issues when responding to direct questions. To some degree, that's a competitive response; the final packaging and pricing decisions are rarely made until quite late in a project's life cycle. It will probably work the way most other similar decisions at Microsoft do: at some point Steve Ballmer will simply tell everyone what the different boxes should contain and how much they'll sell for.

One difficult question I confronted as I developed this book was how much introduction to the underlying hardware (the Intel 386 processor) and software (Windows itself) to provide. Some authors expect you to read other tomes as prerequisites to their own. Still others try to teach you hexadecimal arithmetic before presenting the intimate details of fault-tolerant system design. In the end I decided to support this book's mission by including the information I would need to refer to while talking about the more advanced details of Windows 95. Chapters Two and Three therefore provide a basic description of the Intel 386 processor architecture and the Windows system architecture. If you know these subjects intimately, you can skim quickly through those two chapters. If you never knew much about those subjects, the two chapters should equip you to deal with the new information about Windows 95 in the rest of the book. If you're like me and can't always remember exactly how the 386 paging mechanism works, or precisely what a Windows task *really* is, Chapters Two and Three can serve as a close at hand reference to Intel and Windows architecture.

Windows 95 is built on an operating system base that adds major new capabilities to the system. Some of these new features, such as the new filesystem, have already appeared in other Microsoft operating system products, notably Windows NT and Windows for Workgroups. Windows 95 integrates these new features and other features to provide a full 32-bit protected mode environment for Windows applications. And although MS-DOS compatibility is retained, there really isn't a collection of files in Windows 95 that you can point to and label as MS-DOS. Windows 95 really is a complete operating system for the very first time in the history of this product line. In Chapter Four we'll explore the inner workings of the Windows 95 operating system base.

Every user of Windows will see a dramatic revision in the on-screen appearance of the operating system. In addition to revising the appearance of Windows, Microsoft has changed many of the interactive

procedures and added a unified system control application. In Chapter Five we'll analyze the user interface and the new system shell. That chapter contains a lot of screen shots illustrating various aspects of the shell, and I'm quite sure that some of the details of these screens will change in the final product. I already know that the visuals for the Start menu are a little different, and the "now it's there, now it's gone" game continues with the shell's trashcan: post–Beta-1, the trashcan was back in the product.

Windows 95 introduces some significant changes in both the Windows graphical subsystem and the Windows implementation of device support. For the first time a Microsoft Windows system takes on the challenge of device-independent color—a feature that has become critical to many graphical applications. A major improvement in the architecture for display drivers is also a highlight of the new system-level features you'll see in Windows 95. In Chapter Six we'll take a look at all of these changes.

The architecture for supporting disk devices and their associated filesystems has also changed considerably in Windows 95. A layered device architecture derived from the Windows NT design provides full protected mode support for hard and floppy disks and CD ROM devices. And integrating support for new disk devices into the system becomes comparatively trivial in Windows 95. Although Windows 95 continues to use the MS-DOS FAT filesystem as its default storage scheme, the design of the new installable filesystem manager opens the door for improved filesystem support in the future. Right now, the most visible enhancement in the Windows 95 filesystem is its support for long filenames—finally relieving us of the tiresome 8.3 filenaming convention that has dogged us since 1981. In Chapter Seven we'll inspect the new filesystem design.

Although not limited to operation in the Windows environment, Microsoft's Plug and Play technology makes its system debut with Windows 95. Fully implemented, Plug and Play makes the task of configuring and managing a complex PC a trivial one. Apple Computer won't be able to run those Windows commercials any longer. In Chapter Eight we'll explore the need for Plug and Play and its implementation under Windows 95. Plug and Play capable systems have a life outside Windows 95, and I fully expect Plug and Play systems to be a highlight of this year's COMDEX/Fall trade show. The Plug and Play technology really does work, and if you spend a lot of time messing around with computers, you'll find the benefits of Plug and Play to be compelling—

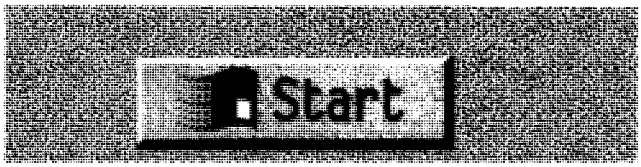
so much so, that I'd recommend your waiting to buy a Plug and Play system as your next.

Windows 95 integrates its support for network systems into the new filesystem architecture. Windows 95 will support several simultaneously active networks—each with multiple connections—and provide consistent interfaces to any underlying network for applications. Some of these features were seen for the first time with the release of Microsoft Windows for Workgroups version 3.11 in the fall of 1993. In Chapter Nine we'll examine network support in Windows 95. The sudden surge of popular interest in the Internet prompted Microsoft to include Internet access utilities in Windows 95 quite late in the development project. It seems likely that the "Internet readiness" of Windows 95 will be a focus of at least some of the early marketing for the product.

Microsoft intends Windows 95 to play a significant role in the growing mobile computing market. Windows 95 features related to that market range from integrated support for pen-based computers to an enormously improved remote network access capability and support for the use of laptop docking systems. In Chapter Ten we'll consider these features together under the general topic of mobile computing. Windows 95 will include support for pen input devices and the associated "inking" operations. Unfortunately, that topic didn't make it into this book—publishing deadlines are a little more rigid than software development deadlines.

Apart from the pen computing capabilities, the only other major feature of Windows 95 that is not a topic of this book is multimedia support. It will be there in the product, but even as late as the spring of 1994 its precise architecture and features were still rather vague. Microsoft seemed to think it was pretty significant that there is a version of the popular Doom game running under Windows using the newly announced WinG graphics library. Game products are really the final bastion of MS-DOS–specific software. Whether Windows 95 multimedia support will be good enough to conquer the games market remains to be seen.

There are components of Windows 95 that will have been in development for well over three years by the time you can go out and buy the product in a store. The first order of business is to look at what Microsoft has been trying to achieve in all that time.



CHAPTER ONE

THE ROAD TO CHICAGO

Throughout its design and development, Microsoft Windows 95 had the codename "Chicago," and the introductory slide for early product presentations depicted a map of the USA entitled "Driving Towards Chicago...." Windows 95 was not designed and developed in a vacuum—there were a lot of stops on the way to Chicago. Beginning with the first release of Windows in November 1985 and continuing through the spectacularly successful introduction of Windows 3.0 in May 1990 and beyond, Microsoft's total investment in Windows has been enormous. Until version 3.0, the commercial returns hardly merited the investment. But no one has ever accused Microsoft of giving up easily, and Windows slowly and steadily improved in both capabilities and sales. The introduction of Windows 3.0 was a watershed event. It was as if the world had suddenly discovered the benefits of Windows, and versions 3.0 and 3.1 sold in great numbers.

In truth, a number of factors contributed to the seemingly sudden success of Windows 3.0. Personal computers using the Intel 386 chip were then becoming affordable. By the time Windows 3.1 was released, 386 systems were commonplace and cheap. The 386 systems provided good performance and the best platform for Windows to run on. Equally as important, the amount of system memory and the quality and performance of video hardware finally matched the requirements set by Windows. Given the now adequate level of system performance, the real benefits of the graphical user interface became apparent to large numbers of users.

Microsoft had long extolled the benefits of Windows, but only a limited number of high-quality Windows-based applications were available before version 3.0. Virtually every demonstration of Windows included Microsoft Excel, Aldus PageMaker, and very little else. There were occasions when Microsoft's own applications development group

questioned the wisdom of pinning all their hopes on Windows, and there were many internal debates, both formal and informal, over the relative priorities of MS-DOS, Windows, UNIX, and OS/2 as application platforms. Windows 3.0 changed every company's perspective significantly, and within several months of its release, the level of application support for Windows had grown dramatically. Software developers were no longer faced with the question of whether it was worthwhile to develop a Windows version of their application—it was simply a question of how fast they could get the Windows version to market.

Even industry journals that had relegated Windows to the also-ran category changed their view. As the numbers of users converting to Windows rose, so did the level of press coverage. Within two years, reviews and discussion of MS-DOS–based products had become the minor news items, and new journals concerning themselves only with Windows had begun to take up a significant amount of magazine rack space.

It was on this stage that Windows 95 would be introduced. Before version 3.0, new releases of Windows had received some polite (and a lot of impolite) interest and had earned the product a few new customers. After all, those were the days when OS/2 had been designated "the next big thing." In that context, Windows version 3.0 was an overachiever, surprising everyone with its improved features and popular success. Microsoft released version 3.1 primarily to solve the problems that widespread use of the 3.0 product had exposed.[1] The product team knew that the stage would be different for the introduction of Windows 95. Expectations were high. Every feature and nuance of the product was certain to be exhaustively examined, discussed, and criticized.[2] Windows 95 had to be the best version of Windows ever, and the goals the team set for the product had to address the need to incorporate dramatic and worthwhile improvements. With sales of the current version of Windows topping a million copies a month by mid-1993, any new release of the product also needed to be totally reliable.

1. Foremost among these problems was the infamous UAE—the Unrecoverable Application Error. Although UAEs were most often caused by bugs in application programs, everyone blamed Windows for UAEs. Eliminating UAEs was the driving motive behind the development of Windows 3.1.

2. One illustration of this high degree of interest: within two weeks of Microsoft's first, limited, external release of the beta, someone had (illegally) provided a copy to *PC Week*. They promptly published a review of the beta—almost a year in advance of the product's planned release date.

Thus, the general goals for Windows 95 were set: build a great new product that includes compelling new features and that is totally reliable—and, of course, develop it quickly. If you've ever worked on a software development project, you probably recognize those grand goals. And you know that every project team has to reduce those nebulous aims to specific targets. With Windows 95, it was no different.

The Mission for Windows 95

Although the goal is expressed in different ways and set in different contexts, one phrase summarizes the mission of the Windows 95 development team: make it easy. The mission to make every aspect of the PC running Windows 95 easier for users, support staff, hardware manufacturers, and software developers consistently reasserts itself. The project mantra often added a qualifying phrase: make it easy, not just easier. Throughout the design and development cycle, each aspect of Windows 95 had to undergo scrutiny within the "make it easy" context.

Help for the End User

Ease of use is an overused phrase in the computer industry. Not that many people find computers easy to use. Most people find Windows easier to use than MS-DOS, but the Windows 95 team recognized that on an absolute scale there was a lot left to do before using Windows would become "easy." These are some of the problems the team recognized:

- Many users remain intimidated by computers. Many potential customers won't buy a PC for the same reason.
- Common tasks, such as setting up a printer, are still far too arduous and error-prone for many users.
- Carrying out a complex operation, such as remote data access, is difficult for sophisticated users and close to impossible for most other people.

The scope for the team's mission also needed broadening. It would be no good making Windows easy to use if the systems on which it ran remained difficult to set up and configure. And Windows 95 itself had to be easy to install and support. To make things easy for the end user at the expense of the MIS department would be self-defeating.

Hardware Platforms

The basic architecture of today's average PC is that of an IBM PC AT–compatible machine, circa 1984. Despite many innovations in components, the overall system design has remained largely unimproved. Beyond encouraging manufacturers to ship PCs with at least a 386SX processor, 4 MB of RAM, and good video boards, Microsoft had done very little in the way of systematically persuading hardware companies to innovate.

Microsoft saw Windows 95 as an opportunity to change the status quo to the benefit of both the end user and the system manufacturer. Central to this effort was the development of the hardware Plug and Play specification, prepared jointly by Microsoft, Intel, Phoenix Technologies (the BIOS suppliers), and Compaq, among others. Plug and Play aimed to eliminate most of the problems associated with setting up and configuring PC hardware. No longer would the user need to know, for instance, what an IRQ or an I/O port address was. The users, their support staffs, and the system suppliers would all benefit from the improved ease of system setup.

Microsoft's other major step to encourage renewed hardware innovation was the decision to finally remove Windows reliance on MS-DOS as its underlying operating system. Successive releases of Windows had incorporated more and more operating system functions, and MS-DOS gradually came to be used as little more than a rather inefficient disk filing system. This trend culminates in Windows 95—a complete operating system implementation that incorporates all the features required of a fully protected 32-bit multitasking operating system. The user needs only to install Windows 95 on the machine; MS-DOS doesn't have to be present on the system at all. Windows 95 continues to support MS-DOS applications using a compatibility feature that has its roots in Microsoft Windows/386, Microsoft OS/2, and Windows NT.[3]

Windows 95 offers the system manufacturer the opportunity to produce improved hardware that doesn't have to conform strictly to the old IBM PC AT design. Such improvements include the incorporation of an improved BIOS and plug-in cards that cooperate with the operating system during system setup. Since device driver software always controls access to any hardware within a Windows 95 system, the user can add any new device provided it has a Windows device driver.

3. Although no code is repeated, members of the Windows 95 team had accumulated a significant amount of expertise when they had implemented similar compatibility features for these other operating systems.

The need for older-style BIOS compatibility no longer exists unless the device must also support MS-DOS operations.

For the Developer—32 Bits at Last

Although the mission statement for Windows 95 emphasized making it easy for users, support staff, and manufacturers, the lifeblood of Windows is still application programs. Early on in life, Windows gathered support from application developers slowly. After the introduction of Windows 3.0, that trickle of support grew into a veritable torrent of new applications. But developing a Windows application was never an easy task, although the quality and variety of development tools and training material have improved by leaps and bounds over those of a few years before. Windows 95 support for 32-bit programs helps the developer significantly:

- Developing 32-bit programs is just plain easier than developing for the 16-bit segmented model required by earlier versions of Windows.
- The Windows 95 32-bit API is compatible with the API supported by Microsoft Windows NT. Developers who want to produce products for both operating systems have an easier time developing and supporting their applications.
- Windows 95 itself uses a 32-bit memory model, and many of the limits of earlier versions of Windows disappear as a result. Valuable system resources, such as file handles, are plentiful. Application developers no longer have to come up with clever schemes to minimize their demands upon the system.

Naturally, the availability and quality of applications for the new release will help determine the success of Windows 95. At the same time that Microsoft worked on Windows 95, they expended even more effort on the development of Windows NT and associated products such as the Advanced Server version of Windows NT. Further mystifying the choice of platforms available to the application developer was word of yet another Microsoft operating system—code-named Cairo—which began to circulate in late 1992.[4] Today the success of each of

4. Chicago's project codename was originally "Tripoli"—a city "very close to Cairo." Humorists on the Windows team then asserted that the name ought to be "Spokane"—a place not very far from Microsoft's headquarters in Redmond. Eventually, "Chicago" was chosen—more because that was the site of the Windows 3.1 introduction than for any other geographic significance.

these operating systems remains undetermined, but before going further along the road to Chicago, we'll look at how Microsoft sees the role of each product over the next few years.

Shall We Go to Chicago or Cairo?

Over the last few years, every one of us has had several opportunities to change PC operating systems. The sheer size of the installed base of MS-DOS systems and application software creates enormous inertia, and with no compelling reason to change, people simply don't. This hasn't stopped a variety of vendors from trying to replace MS-DOS with a better mousetrap. UNIX, for example, in all its versions, has been around even longer than MS-DOS, and each year brings a renewed pledge of unity and coherence from the UNIX vendors. Usually the vendor infighting reasserts itself about six months later, and UNIX returns to its status of technical overachiever and commercial also-ran.

Microsoft, in partnership with IBM, tried to replace MS-DOS with OS/2. After a few years and tens of millions of dollars spent in development and promotion, OS/2 was nowhere in the market. Microsoft abandoned its OS/2 efforts shortly after the introduction of Windows version 3.0, when it became clear that Windows would be very successful and OS/2 would never be a good enough product to justify a switch from MS-DOS. Microsoft did press on with the development of another advanced operating system, however—Windows NT. Why? Hadn't enough money been wasted on trying to replace MS-DOS? Wouldn't it have been better just to improve MS-DOS itself?

Technically speaking, MS-DOS is a severely limited operating system. Its inability to support proper multitasking, memory protection, and large address spaces makes it a poor base for environments where the user wants to run several complex applications while connected to a network. Fixing these problems involves much more than making modifications to MS-DOS—it really does take a new operating system. To a degree, Microsoft was able to incorporate some necessary improvements to MS-DOS into successive versions of Windows. Multitasking, limited 32-bit application support, memory protection, and other features are now all functions of the current release of Windows. This way of evolving an operating system also passes the test for commercial rationality. Since Windows required MS-DOS to be on the system already, it was easy for users to upgrade, and Microsoft could add new functions without having to change MS-DOS itself. In fact, by the time Windows

version 3.1 appeared, Windows used MS-DOS for not much more than loading programs and managing the disk filesystem.

First Stop—Chicago

Windows 95 is a major step in an evolutionary process. On a system running Windows 95, there is no longer any need for a separate product called MS-DOS. Windows 95 takes over all the operating system functions. You install a single product, and when you boot the system, you go directly into the Windows environment. You'll no longer see the familiar C:> prompt at which you typed the *win* command. Naturally, Windows 95 retains MS-DOS compatibility so that you can still run all of your existing TSR programs and any other MS-DOS applications you use. But the basic architecture of Windows 95 is Windows with MS-DOS compatibility. It is not MS-DOS running a Windows subsystem.

There are a lot of technical reasons for implementing Windows 95 this way. Relying at all on MS-DOS as the basic operating system would have reduced the capability and performance of the overall system. Now Windows truly supports the functions needed for advanced applications and networked systems.

This evolutionary progression in the architecture was also feasible from a marketing perspective. When Windows wasn't very popular, it would have been impossible to persuade people to give up MS-DOS and move to an alternative. This conversion is exactly what the OS/2 campaign failed to pull off. Now Windows is popular, and users spend much more time running Windows applications than they do MS-DOS applications. Thus, Windows 95 is a great upgrade to Windows 3.1, and yes, you can still run those aging MS-DOS applications.[5]

At this point, you might be wondering whether Microsoft is once again predicting the imminent demise of MS-DOS. Probably not. There is an active MS-DOS development group at Microsoft, and MS-DOS versions 5.0, 6.0, and now 6.22 attest to their efforts. The possibility of the protected mode operating system components of Windows 95 forming the basis of an MS-DOS 7.0 release was the subject of much questioning and speculation during 1993. Microsoft would not confirm the speculation, at least not by July 1994, but it's impossible to ignore the commercial success of the retail upgrade packages for MS-DOS 5.0 and 6.0. An MS-DOS 7.0 upgrade release could provide both significant user benefit and plenty of revenue dollars.

5. Demonstrating their personal bias quite succinctly, Microsoft executives referred to the release of WordPerfect 6.0 for MS-DOS as "the last great DOS application."

Clients and Servers

Apart from the move to Windows, the other major trend over the last few years has been the widespread adoption of high-speed local area networks. Sometimes these LANs have been installed where there were no computers before, and now they are often installed to replace mainframe- and minicomputer-based systems. Each machine on the network usually operates in one of two roles: as a client (typically the system that's on your desk running your applications) or as a server (where the systemwide databases and other shared resources, such as printers, are found).

For a client system, you need a high level of usability, great graphical display performance, and an easy to manage network connection. Some newer machines, such as the smallest portable systems, probably spend a lot of their time not connected to anything. At some point, though, even they have to become true clients, perhaps to print a file or to connect to an electronic mail network.

For a server, you need performance, performance, performance, and, of course, performance. Actually, the modern PC network server needs to offer a lot of complex features:

- Performance. The server operating system must be very efficient at transferring data across the network. To meet the performance demand, the operating system must also support machines using multiple processors, very high speed, high capacity disk drives, and high-performance network hardware.
- Robustness. This word means that the system doesn't crash and that if it does, it doesn't destroy data in the process. This requirement extends to the operating system's ability to protect different programs from each other's weaknesses. If your wide area communications server falls over in a heap, for example, you'd certainly prefer that it didn't take the database server down with it.
- Security. Securing data has always been a concern for any computer system that many people can access, whether the access be by virtue of proximity or through incoming telephone lines. Research efforts in the last few years have formalized many aspects of data security, and modern operating systems are expected to meet some specific requirements. Most governments insist that computer systems meet demonstrated, and certified, security standards, and many corporations have adopted a corresponding policy.

- Network management. If you have a large network that is geographically dispersed, you need the software tools that allow you to manage it effectively. Activities might range from simple tasks, such as adding and removing network printers, to finding and updating every copy of a particular application program throughout the network.
- Transparent distribution of data and processing power. Ideally, a network system should allow the user to retrieve data and access other resources without having to know the network locations of the objects in question. Although your client desktop system participates in locating and using resources, it's the server that has to figure out where a resource is and how to give you the most efficient access to it.

Of course, you'd like all these server features on your client machine as well. Unfortunately, implementing these advanced capabilities takes a lot of software, and that translates into the need for more memory, more disk space, and more processor speed. Someday we'll all have 500-MHz processors with gigabytes of memory in our laptop machines and we'll install the most powerful version of everything. Of course, by then, we'll have figured out some new feature that we simply must have and for which we still won't have enough hardware capacity. Until then, the configuration of most desktop and portable machines is likely to be a lot smaller and cheaper than a server configuration. Operating system vendors generally target a particular product toward either the client-type machine or the server machine.

Microsoft's operating system development efforts acknowledge the differences between these two basic system types. For the high-volume client-type machine, Windows 95 is the product Microsoft wants you to use. As we'll see when we look at the features of Windows 95, there is a very close mapping between its features and user requirements within the client market segment.[6]

The lowest-power machine configuration the Windows 95 team had in mind was an Intel 386SX–based system with 4 MB of memory, a VGA display, and 80 MB of disk space. In 1994, that's a pretty simple and cheap configuration. But Windows 95 had to run at least as well as Windows 3.1 on such a system. The Windows 95 team didn't try to implement the complex security features or multiprocessor support offered

6. Another early Windows 95 marketing slogan—every Microsoft product accumulates many before the final tagline is chosen—was "the ideal client system."

by Windows NT.[7] Such features would have added a lot to the operating system's hardware requirements, and most users simply don't need or want such features. Certainly for the portable computer market, which represents a large share of potential Windows 95 sales, such features are neither applicable nor even desirable.

For the server market, Microsoft says choose Windows NT. With Windows NT, you'll get virtually unlimited capacity and the features that meet all of the server requirements we've just looked at. Many users will have computing requirements that demand the capabilities of a Windows NT machine right there on the desktop. Their work will also justify the use of a machine with the power of an Intel 486, 16 MB of memory, and 256 MB of disk space. Today that's still a pretty impressive configuration for a desktop machine, but for a network server it's not much more than an entry-level configuration. Of course, the incredible pace of improvement in personal computer hardware will make that 486 configuration a low-end system within a couple of years, and users will be able to choose to move up to Windows NT functionality with no loss of performance.[8]

And On to Cairo

The first thing to note about Cairo is that its new features don't make up a complete operating system. Cairo will actually appear as Microsoft Windows NT version something point something. Windows NT will continue as the base operating system, performing all the memory management, task management, device handling, printing, and so on. In some ways, this arrangement is similar to the way in which successive releases of Windows before Windows 95 added new capabilities to the MS-DOS operating system. For Cairo, however, the underlying operating system is an immensely powerful one. Microsoft freely acknowledges that in the first release of Windows NT it sacrificed advances in usability to designing and building an operating system with a sophisticated and long-lived architecture. Cairo seeks to augment the native capabilities of Windows NT rather than add features that should be in the operating system proper.

7. Windows NT also runs on processors other than the Intel 80386/486/Pentium family. This portability was never a goal for Windows 95. The enormous difficulty of maintaining full MS-DOS and Windows compatibility, let alone the implementation effort that would be needed, made this idea a non-starter.

8. Remember that it was only early 1988 when the very first 16-MHz 386 machines with 4 MB of memory were considered to be high-end systems.

If you plan to use Windows 95, then, in a sense you'll use the first incarnation of Cairo. In particular, the new look of the Windows 95 interface and of the system shell will appear in Cairo too.[9] There will be a lot more to Cairo than the new look, of course, but as far as appearance is concerned, you'll be immediately familiar with the product. Cairo will be a completely object-oriented system, allowing you to query networkwide for a data object and examine it as you choose. Cairo will make it easy for you to query the network for all the memos authored by people in your department, for example. You won't need to know anything about filenames, filename extensions, what servers might contain the document files, and so forth. If your network administrator increases capacity by adding a new network server and splitting the data between the old and new servers, Cairo will keep track of what happened. You'll formulate your next query and get the results oblivious to the fact that a configuration change has occurred.

No doubt you're wondering how much hardware power will be necessary to run Cairo effectively. No doubt a lot. No doubt you'll need a machine that today would be considered only for duty as a network server. But by the time Cairo comes up for adoption as the mainstream operating system, that amount of computing power will be available in a reasonably priced desktop machine. Someday microprocessor engineers may reach an absolute physical limit, but that seems likely to be a day that you and I won't much care about.

So what of Windows 95 in this networked world? Microsoft plans to extend the Windows role as the perfect client-side operating system and to ensure its continued suitability for less powerful hardware, portable machines, and pen-based systems—few of which will run Cairo. Through an update to Windows 95, Microsoft will make available the tools that client systems will need to access Cairo systems effectively. You'll use your Windows machine to formulate queries, for example, but it will be the Cairo systems that take care of searching the network and retrieving the information. Application programs designed for the Cairo environment will exist as distributed applications. Part of the software will run on the Windows machine and communicate with a server-side application running somewhere else on the network.

9. A lot of the original design for the new user interface was actually done by people on the Cairo team. It was up to the Windows 95 group to implement the interface and bring it to market, but there was an ongoing effort to ensure consistency with the evolving Cairo design.

Summary

During 1993 Microsoft began the usual seeding process that precedes all of their major product releases. The company repeated its intention to build Windows into a family of compatible operating systems that would cover market requirements from mission-critical corporate computing to consumer devices. The executives who gave the public presentations used the slide shown in Figure 1-1 to illustrate their view of the evolution of the Windows family.[10]

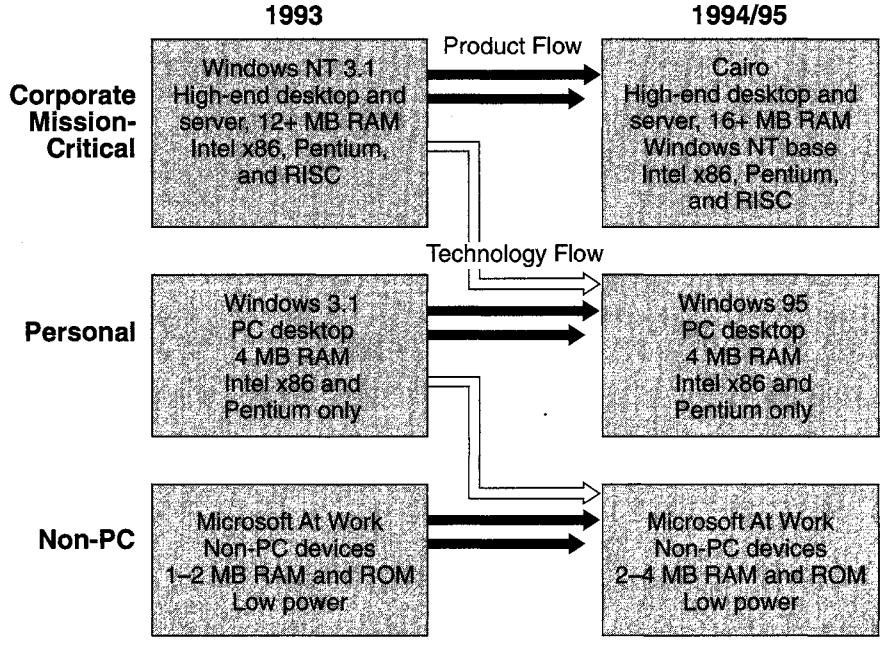


Figure 1-1.
Evolution of the Windows operating system family.

As you can see, a coherent story underlies all the different products. The products evolve in capability, and features can migrate to other operating systems as microcomputer technology allows. Microsoft itself is a firm believer in the continuing growth of microprocessor capability. This increase in horsepower is largely what allows the advanced features of, say, Windows NT version 3.1 to appear in other operating systems.

10. The form of this slide changed over time, but the basic message remained the same.

Whether Cairo will be successful is a question that can't be answered for a few years yet, since its story will be played out much further into the future than the Windows 95 story. Let's get back to our main subject and take a detailed look at what the Windows 95 team set out to do.

Project Goals

Let's review the market context for Windows 95:

- Windows 95 would be the next release of an immensely popular product, Windows 3.1.
- A huge amount of installed software, both for MS-DOS and for Windows, placed some stringent compatibility requirements on Windows 95.
- There was a real desire on Microsoft's part to make Windows 95 easier to set up, use, and administer.
- There was a need, principally for the benefit of Windows application developers, to dramatically improve the fundamental capabilities of the system. More resource and memory capacity, better performance, and support for more complex programs appeared at the top of most petitioners' lists.
- Windows 3.1 appeared in mid-1992. Obviously the next version of Windows had to make it to market in a reasonable amount of time after that—meaning that 1997 wouldn't cut it.
- Other operating system development projects were proceeding in parallel at Microsoft. Care had to be taken to ensure compatibility with both Windows NT and the Cairo efforts and with the release of Windows for Workgroups 3.11 in November 1993.

From the very early discussions about what the Windows 95 product should be, there emerged a specification that translated these loose market requirements into a more precise statement of goals for the project. Each section of the more detailed specification addressed these ten issues almost as ten commandments and described how each particular feature met the basic project goals.[11] The specification grouped

11. By the time work on this book began in earnest in April 1993, the *Chicago Feature Specification* was approaching its eighth substantial revision and stretched to over 200 densely printed pages. Who said software was all about writing tight code?

the ten issues as "The Four Requirements" and "The Six Areas for Improvement." By and large, these ten goals remained unchanged during the development project.[12] Here's how the feature specification summarized them (verbatim):

The four requirements:

- Compatibility
- Performance equal to or better than Windows 3.1 performance on a 4-MB system
- Robustness
- Product availability in mid-1994

The six areas for improvement:

- Great setup and easy configuration (Plug and Play)
- New shell and user interface visuals
- Integrated and complete protect mode operating system
- Great network client, peer server, and workgroup functionality
- Great mobile computing environment
- Windows 32-bit application support

A lot of this book is a detailed examination of the major new features of Windows 95. Before launching into the detail, it's worth taking a brief look at what these project goals really mean.

Compatibility

Compatibility is both the dream and the nightmare of everyone who develops products for the PC market. The basic PC architecture was defined by IBM's very first product introduction in August 1981. Once the clone (later "industry standard") manufacturers were established and software developers had figured out what compatibility meant for them, the industry grew spectacularly. Compatibility means that you and I can walk into a computer store, buy any PC product there, install

12. The original requirements specified "great 4-megabyte system" and "product availability in the first half of 1994." As you can see, the performance goal became more precise and the availability goal extended beyond its upper bound.

it, and expect it to work. Great news for us. Unfortunately for the developers of PC hardware and software, compatibility means that you and I can walk into a computer store, buy any PC product there, install it, and expect it to work. Any developer has to do a certain amount of compatibility testing before releasing a product. For a straightforward application program, the developer's testing problem is a finite one that might only involve testing on popular networks and with popular printers. For a more complex product, such as a memory resident communications program that runs in the background, the testing matrix becomes much larger. The development effort could involve testing for compatibility with different networks, different modems, and different versions of MS-DOS, PC-DOS, DR-DOS, and Windows, with other memory resident programs, ad infinitum. This testing burden represents a substantial part of the product's development cost.

Now consider Windows 95. For the product to be successful, it simply had to be compatible with everything that had gone before—not only Windows applications software, but MS-DOS applications, device driver software, and network software, to name the principal foes. If the product were truly compatible, the reasoning went, the new features alone would persuade every user to upgrade without a second thought.[13] And the absence of a "real" MS-DOS in the Windows 95 architecture was a radical revision that seemed guaranteed to produce some difficult to solve compatibility issues. Clearly, Windows 95 needed a massive compatibility test effort, and that's what the Windows 95 team set about organizing.

The Compatibility Fallback

Microsoft also decided that Windows 95 needed an ultimate compatibility fallback. Everyone was sure that the fallback would be invoked only in the event the user wanted to run some ancient, obscure game software. But the fallback did represent a good insurance policy against any case in which Windows 95 broke the compatibility regime.

The fallback solution is to allow the user to exit completely from Windows and run an actual real mode MS-DOS. While the system runs in this mode, a small software loader stays resident in memory. That's the only component of Windows 95 still memory resident while the system is in MS-DOS real mode. Once the user finishes off the Klingon empire, the software loader traps the application program's exit call and reloads Windows from disk, returning the system to its normal state.

13. Referred to in Microsoft vernacular as a "no brainer upgrade."

Performance

The earlier versions of Windows garnered a healthy measure of criticism on several fronts. Poor performance was an oft-repeated complaint. Looking back at the hardware configurations then available for Windows, it seems amazing that the product was even usable. In 1985, Windows was able to run on a 286-based system with a poor display adapter (the CGA), a single megabyte of memory, and a fairly slow hard disk. Any popular laptop system today has a comparatively much improved display and better disk hardware, four times as much memory, and a processor probably 25 or 30 times faster than the first 286. Naturally, Windows has obeyed one of the unwritten laws of computer science and expanded to consume all the available hardware resources.

It's hard to measure the performance of a Windows system in absolute terms. Does a benchmark reading of 15 million Winmarks mean that you'll see your desktop publishing package run at lightning speed? Generally, users will judge a product's performance from its response time. Snappy screen redrawing, fast file opening and closing, and quick scrolling operations always make a good impression. Less easy to observe but equally important to the overall system performance are operations like network data transfers and program swapping. The operating system vendor thus has to invest in two parallel performance measuring activities: checking individual operations, such as how fast a program can read a file, and observing the whole system as it runs a mixture of applications and data transfer operations.

Microsoft's development teams have always focused on performance issues. They tune individual software components for improved speed and reduced memory consumption as well as raise overall system performance by removing undesirable interactions among different components. Within Windows 95 itself, new features such as the 32-bit protected mode filesystem and dynamically loadable device drivers were aimed at improving system performance. Would the end user like to see the system run even faster? Of course, but the recent performance of Windows 3.1 on the base configuration 386SX with 4 MB of memory is generally considered as reasonable.

For Windows 95, the development group set itself the goal of running as well as or better than Windows 3.1 on the same base hardware configuration. Not very ambitious, you might say. However, this goal took into account that the system had to include the new capabilities such as the Plug and Play subsystem with its dynamic reconfiguration facility at the same time that it ran the application mix. Adding significant

functionality while maintaining the same level of performance is ambitious. By simple extension, a Windows 95 system doing exactly what the Windows 3.1 system did, on the same hardware, ought to run faster. Measuring different application mixes, modeling end user activities, and playing with the variables have been staple ingredients of Windows 95 performance analysis.

The key, repeated phrase in Microsoft's later Windows 95 presentations was "as well as Windows 3.1." The recurrence of this phrase emphasized the fact that Windows 3.1 on a 4-MB system running Microsoft Office and using OLE performs dreadfully. The Windows 95 team didn't try to address this problem. In fairness, they couldn't. An application mix of this complexity demands more memory—at least 8 MB and probably more. Fortunately, early 1994 saw 8 MB becoming the default configuration for many machines, so, to some degree, the problem would be solved by the time Windows 95 was released.

In early 1994, performance tuning began in earnest, and all of the project status reports for Windows 95 dwelt on performance tuning issues for some months. By the time of the Beta-1 release, Windows 95 performance was already as good as or better than Windows 3.1 performance in almost every respect.

Robustness—Adieu UAE?

A robust system is a system that doesn't crash—whatever the user or application programs do to it. If one program goes awry, the user can halt it without affecting the operation of any other programs or losing any data. If a program makes erroneous requests for operating system services, the system protects itself by terminating the offending program with no effect on other programs.

Windows 3.0 was roundly criticized for system crashes. The infamous unrecoverable application error (UAE) was a widely publicized, and poorly understood, problem. Windows 3.0 reported a UAE whenever it determined that the system itself had reached an inconsistent state. An application used a file handle to access a file that had been deleted, for example. For most of the UAEs, the error was actually in the application program and not in Windows itself. However, Windows 3.0 did a poor job of validating system requests generated by application programs. Thus, an application could make an invalid request that Windows happily accepted and tried to process. By the time the error was discovered, there would be nothing left to do but crash the system as a rather primitive last line of defense. Fixing this problem was a focus of the work to produce Windows 3.1, which carefully validated almost every

system request before processing it. As a result, many application vendors had to release updates of their products to fix software bugs that had never been discovered before. The experience was a painful one for all concerned, and the Windows 95 team was in no rush to repeat it.

The development team wanted Windows 95 to be extremely robust, with almost no possibility of a system crash caused by an application program or other external factor. How do you go about ensuring this? A lot of the answer goes back to the basic design of the system: incorporating careful validation of application requests, protecting system data regions, and isolating individual software components. In particular, the new 32-bit application programming model allowed the Windows 95 team to implement full memory protection for individual 32-bit programs. Not only are 32-bit programs protected from each other, but the system is also fully protected from these programs. (Some improvements were also made for 16-bit programs, but the options were limited because of compatibility constraints.) Once all of this is done, you test and test and test some more.

Timely Product Availability

The eternal battle between the sales and marketing group and the development group within any software project comes down to deciding when the product is ready for release. Microsoft always sets an estimated release date for a product way ahead of detailed planning. Then the development team either cuts features or extends the planned release date to allow completion of all the development work. Factors that influence the release date include when the previous version was released, the overall scope of the work for the new version, and how competitive the market is. The decision to bless a particular version of the software as the "golden master"[14] involves many people from the product group, senior managers within the development division, product support personnel, and often Bill Gates himself. If the product is simply not ready for release because of performance inadequacies or major bugs, there's no debate—you slip the date, and the development team continues its work. But there finally comes a point when the software is in good shape, all the introduction materials are ready, the support personnel are trained, and the printed documentation is waiting in the warehouse.

14. In Microsoft parlance, the development group prepares a succession of "release candidates" before shipment. When everyone is satisfied with the quality of the software, the final release candidate becomes the golden master from which the manufacturing group prepares the production version.

There are still some bugs that could be fixed if you were to wake up the development team and get them to put in yet another day or another week of effort. Do you ship the software or do you wait? In any complex software product, from any company, bugs always remain in the shipping version. Experience and judgment dictate when those bugs are sufficiently unobtrusive that the software really is ready for shipment.

Windows 95 has been no different in this respect. By the middle of 1993, Microsoft had come up with the product's original, and rather vague, shipment goal of "the first half of 1994." This date would be about two years after the release of Windows version 3.1, and that was one major factor in choosing the planned ship date for Windows 95. Once the scope of the work was better understood, the development team pinned the release date down more firmly to "mid-1994." Plans were also made for a succession of limited releases to software developers, beta test sites, and others before the final general release. This cycle of controlled releases began in August 1993, almost a year before the planned general release date. The fact that a pretty complete and functional version of Windows 95 was available that early on says a lot about the extent of the testing and improvement effort Microsoft planned for the product before it would release the final version.

Well, guess what? The team completely blew the mid-1994 date. In fact, the Beta-1 release barely made it before the end of June. Once again, it proved to be beyond human ability to accurately forecast the completion date for a complex software project. This difficulty is not unique to Microsoft's release date predictions. Virtually no one is able to forecast with any accuracy, but Microsoft's plans are often very public. The most public statement of the release goal was Bill Gates's speech at the 1994 COMDEX/Spring show, when he demonstrated Windows 95 and committed to a release date of "before the end of the year."

How well "before the end of 1994" will be met remains to be seen. But rest assured that many long workdays and sleepless nights have yet to be invested in Windows 95.

Easy Setup and Configuration

Setting up and configuring a Windows system has never been a trivial task. Each new release has improved the process, but even the setup for Windows 3.0 and 3.1 (considered to have made quantum leaps in this area) has continued to baffle a lot of users. The "make it easy" directive governed much of the effort invested in improvements to the system

setup and configuration procedures. The Windows 95 team decided to concentrate on these areas for major improvement:

- Hardware configuration. The Plug and Play initiative was intended to dramatically ease the process of configuring PCs, and Windows 95 would be the first operating system product to support the Plug and Play standard that Microsoft, Intel, Phoenix Technologies, and others were preparing.
- Installing and configuring Windows 95 on an existing Windows 3.1 system. The team felt that this process ought to require no user involvement beyond swapping diskettes at the right time. After all, if a system ran Windows 3.1, someone must have solved any setup or configuration problems already. Windows 95 ought to be able to use the earlier effort to ease its own installation process.
- System administration and reconfiguration procedures. Every aspect of the existing system was carefully analyzed to improve ease of use. For example, the team felt that any user ought to be able to set up a new printer without a problem. With Windows 3.1, that had not always been the case.

The Plug and Play Initiative

The Plug and Play standard was an effort with a much broader scope than simply Windows 95. Intended by its sponsors to be independent of any particular operating system, Plug and Play defines extensions to the existing PC hardware architecture, together with new BIOS and device driver capabilities that aim to shield the user from hardware setup and configuration issues. Apart from the physical process of plugging a system or a device in and turning it on, Plug and Play takes over the problems of identifying a device, assigning the device the correct hardware configuration resources (such as an interrupt request level), and configuring the appropriate device driver software.

Plug and Play is also independent of any particular bus architecture. It will use ISA, EISA, Micro Channel, PCMCIA, or any other bus architecture that has some market share. In the case of the ISA bus, in which there is really no hardware support for Plug and Play operations, the specification defines a new adapter card interface. For a small additional hardware cost (perhaps 25 or 50 cents) and with some new software, an ISA adapter card can become Plug and Play compliant. For even non-Plug and Play systems, a large amount of effort went into developing

device recognition and configuration capabilities. We'll take a detailed look at the whole Plug and Play architecture in Chapter 8.

Configuring Windows

Configuring Windows itself has become something of a black art. Lengthy articles, and even whole books, devote considerable attention to every one of the often obscure lines in the Windows WIN.INI and SYSTEM.INI files. Coupling the contents of these two files with the contents of the basic CONFIG.SYS and AUTOEXEC.BAT files means that the user trying to modify or improve the operation of Windows faces a daunting task. The Windows 95 team decided to subject every single entry in the configuration files to detailed scrutiny. If an entry really wasn't needed, why was it there? Furthermore, why were there so many special case entries? Could better default selections avoid the need for additional entries? Did Plug and Play make some entries redundant? The more settings that could be eliminated, the easier the system would be to understand.

Apart from the files that control Windows operations, many applications use private initialization files or add parameter information to the WIN.INI file. Rationalizing this whole configuration mess was long overdue, and the Windows 95 team adopted the solution designed by the Windows NT group. Windows NT uses a special file called the *registry* to contain all the information relating to hardware, operating system, and application configuration. Entries in the registry are available to application programs through defined application programming interfaces. Applications can add to and retrieve their private configuration settings using registry access APIs. No longer can the user edit the text in a configuration file and introduce inconsistencies or other errors. Windows 95 uses the registry concept in an identical way, and as developers update application programs for Windows 95, the jumble of configuration files will disappear.

User-Level Operations

Many basic system management operations, such as setting up printers or modifying the layout of the Windows desktop, ought to be available to every user. Yes, they're there, but some of them are awkward to use and difficult to comprehend. Windows 95 addresses this problem by consolidating and simplifying many of the day-to-day operations that all users must perform on their own systems.

New Shell and User Interface

The most immediately striking aspect of Windows 95 is the new look of the screen display. Microsoft uses visual designers on all of its projects these days, and the attention to details of the Windows 95 appearance is remarkable. No longer does a programmer spend a mere hour designing a new icon for a control panel function. The process now involves a visual designer who carefully considers the intent, appearance, and overall consistency of the new visual element. At first glance, there's no obvious difference between individual screen elements of Windows 3.1 and those of Windows 95—no immediately apparent changes in an icon, for example. But if you look closely, you can see the subtle alterations to the shading and the shadow illusion around the icons in the Windows 95 version. As you can imagine, a lot of debate and painstaking effort went into the revision of the appearance of Windows 95. Later in the book, we'll examine these changes in detail.

The New Shell

Much more than just a pretty new face, the Windows 95 shell is a major functional step forward. Asking a Windows 3.1 user to identify "the shell" elicits some interesting responses. Some people have no idea what the shell is. Those who do have an idea will often identify the Program Manager as the shell component. Further questioning about how the File Manager, Print Manager, Task Manager, and Control Panel fit in with "the shell" will usually leave even the most expert Windows user confused.

This confusion is not because the user doesn't understand the system: Windows actually is rather confusing. For example, why do you configure printers using the Control Panel, alter print characteristics using the Print Setup option on the application's File menu, and then control print spooling using the Print Manager? Most proficient Windows users become accustomed to these procedures and forget about the awkwardness, but trying to introduce a naive user to the system and justifying, or even explaining, this scattered approach is difficult.

Fortunately, Microsoft itself recognized the problem a long time ago, and the Windows 95 release represents a serious effort to unify and improve the collection of system functions that form the shell. Of course, there are some major new features beyond that:

- OLE 2 is the first step in Microsoft's initiative to move toward a document-centric application architecture. The Windows 95 shell supports OLE 2 functions and consistent drag and drop capabilities.

- Electronic mail is almost a given in a networked environment. The shell supports an electronic mail interface directly.
- Long filenames—at last you can name a file *My chicken chili recipe* and not have to use CHCHRECP.DOC, ensuring that a month later you won't have the vaguest idea what the file contains.
- File viewers have become popular for allowing a user to examine a formatted file without having to access the application that created the file. Windows 95 incorporates a set of viewers.
- Pen gestures that were originally defined for Microsoft Pen Windows have been revised and incorporated directly into Windows 95. As the base of pen systems expands, Windows 95 will support pen systems without having to add new operating system components.
- MS-DOS applications will most likely live forever. Although Windows 95 appears to hasten their demise by providing a better Windows environment, the support for MS-DOS applications is also improved in Windows 95. MS-DOS window sizing, copy and paste operations, and the use of TrueType fonts within an MS-DOS application are among the improvements.

Complete Protected Mode Operating System

Later on in the book, we'll look at exactly what protected mode is and at what it means to Windows. Suffice it to say at this point that use of the protected mode removes memory limitations—that is, the 640K barrier disappears—and provides a solid basis for ensuring system robustness. The greater part of Windows 3.1 is a protected mode system. MS-DOS itself, however, remains a real mode system. Consequently, a system running Windows 3.1 continually switches back and forth between protected mode and real mode.[15] The switching overhead detracts from system performance.

The decision to implement Windows 95 as a complete system, no longer reliant on MS-DOS, opened the door to dispensing with all the remaining real mode components. In particular, the filesystem (handled by MS-DOS when you run Windows 3.1) and the mouse driver could

15. Actually, virtual 8086 mode—it's not quite as bad as real mode.

now be rewritten as protected mode software. Given the protected mode base and its enhanced capabilities, other improvements were obvious. For example, the print spooler could become a true preemptively scheduled background program. And some of the limitations of the Windows device driver model (the so called VxDs) could be removed, allowing VxDs to be dynamically loaded and unloaded rather than reside permanently in memory as in Windows 3.1.

The other aspect of completeness that the development team planned to tackle was filling in the gaps still present in Windows utility functions. Windows 3.1 has no equivalent to the MS-DOS Chkdsk program, for example. If you want to run the Chkdsk utility, you have to exit Windows to do it. Getting rid of such inconveniences was all part of the goal to provide a complete operating system.

Also on the list of operating system improvements was the removal of redundant and conflicting functions. Windows 3.1 introduced a very successful printing model that incorporated a single major module supplemented by small, simple device-specific printer drivers. This model had a number of positive effects, including the elimination of a lot of duplicate code in the different printer drivers and the promotion of the quick development of new drivers with fewer errors. Windows NT made use of a similar concept to standardize disk device support. Windows 95 would continue along the same path by using a similar model for its hard disk, SCSI device, display, and communications driver support.

32-Bit Application Support

Along with the growth in complexity of modern operating systems and computer networks has come a growth in the depth and breadth of single application programs. No longer does a word processor simply allow you to put words on paper. Customers expect spelling and grammar checking functions, a thesaurus, page layout facilities, and a host of other features. The sheer scope of today's application programs calls for the consumption of large amounts of memory, disk space, and processor cycles. Despite the fact that Intel's first true 32-bit chip began to appear in PCs in 1988, MS-DOS and Windows have never fully supported 32-bit application programs. Rather inadequate solutions, such as the DPMI standard incorporated into Windows 3.0, have been little more than stopgaps to the developers who desperately needed 32 bits' worth of memory addressing.

Windows NT was Microsoft's first operating system in the Windows family to offer full 32-bit support. Windows 95 will join Windows NT in supporting Microsoft's Win32 32-bit application programming interface. From the application developer's point of view, 32-bit support provides three major benefits:

- Access to essentially unlimited amounts of memory. A single Win32 application program can access up to 2 GB of memory.
- A much easier to program memory model. Writing software for a so called "flat," or linear, 32-bit address space provides relief from the vagaries of the Intel processor family's segmented architecture. A programmer can design data structures without having to worry about the boundaries and limitations imposed by a 16-bit memory model.
- A consistent application programming interface. The Windows API contains hundreds of functions that together involve thousands of parameters. In Windows 3.1, some of the parameters are 16 bits and some are 32 bits. It is a rare programmer who can remember which is which and never make mistakes while writing code that calls these APIs. Win32 functions consistently use 32-bit parameters with a consequent reduction in programming errors.

Before the development of Windows 95, Microsoft defined a subset API termed *Win32s*. Included within the Win32s definition were all the APIs that, if strictly adhered to, would allow an application developer to produce software that would run on both 16-bit Windows 3.1 and 32-bit Windows NT. Win32s was in fact a true subset of the Windows NT API and was made available on Windows 3.1 through the use of a library that converted the Win32s 32-bit API calls to the native 16-bit API calls of Windows 3.1.

The Windows 95 team needed to improve on the original Win32s API set and originally defined a *Win32c* API set that took Win32s as its base and added a number of APIs specific to Windows 95. For example, device-independent color capabilities (important in most desktop publishing and drawing programs) will appear for the first time in Windows 95. The term Win32c became quite confusing, quite quickly, and many questions about the relationships among Win32, Win32s, and Win32c convinced Microsoft that they needed a

simpler story.[16] After an interval, the Win32c term was dropped altogether, and the Windows 95 Win32 API set became simply a subset of the full Win32 API, defined (at that time) by Windows NT and slated for expansion in the Cairo era.

The exact definition of the Win32 API set and the individual levels of support in each operating system for the Win32 API can be found only by consulting the appropriate documentation. Microsoft's intention is to allow an application program conforming to the Win32s API to run on any Windows operating system (from Windows 3.1 onward). Applications that use more advanced capabilities cannot necessarily be supported on every version of Windows. For example, applications using the advanced security features available in the Win32 API will run only on Windows NT and its direct successors.

The Jump to 32 Bits

Moving to the 32-bit API under Windows 95 introduces an interesting discontinuity, and for once, discontinuity provides a useful break with the fully compatible past. Since developers who decide to use Win32 must modify their application code, Microsoft reasoned that they could impose a rule on developers requiring that every API in an application be a Win32 API. Thus, not only do you modify your code to incorporate the new 32-bit device-independent color APIs, but you also modify all the other Windows API calls to conform to the Win32 interface. This includes the basic APIs that deal with issues such as file management and memory allocation.[17]

Given this new application model, and its associated rules, the Windows 95 team could incorporate some significant new capabilities into Windows 95. Since the system would know that it was dealing only with applications that conform to the Win32 rules, it would know how to manage the applications a lot more effectively than it could the existing 16-bit applications. Under Windows 95, the benefits realized by an application that bases itself on Win32 extend far beyond simply having 32 bits' worth of memory—notably:

- Preemption. A Win32 application is fully preemptible, meaning that the operating system can suspend its execution at any moment in order to switch to a higher-priority task. In

16. The first interesting marketing sleight of hand simply modified the interpretation of the *c* in Win32c to say that it was for Win32 common, rather than Win32 Chicago. This didn't go far enough, however.

17. To their credit, Microsoft supplied a program analyzer that simplified a lot of the grunt work needed to complete this type of conversion.

general, this means smoother response (an hourglass displayed by one application no longer means that you can't switch to another to do something useful), better system throughput, and avoidance of the data loss that can come from an application's having to wait too long for control of the processor.

- Separate address space. A Win32 application runs within its own protected memory region. No other application can scramble its code or data.
- Thread support. Often a single application would like to do two things at once—perhaps writing a backup copy of the current document to disk while still allowing the user to edit the on-screen text. Under Windows 3.1, multitasking within a simple application is an awkward and error-prone feature to implement. An application's ability under Win32 to utilize multiple threads of execution provides a structured way to perform multitasking.

Networking and Mobile Computing

Microsoft originally introduced its peer-to-peer local area networking extension for Windows in the fall of 1992. Windows 95 essentially incorporates the Windows for Workgroups local area network functionality and thus mirrors the model that Windows NT established. Microsoft has long espoused the belief that networking capability is a fundamental part of the operating system. Separating networking and operating system products into different categories, or using special purpose operating systems for network servers, really isn't the way to go. However, Windows 95 enters a world in which Novell servers make up the major part of the installed base. For Windows 95 to become popular in a Novell-dominated network environment, it needs to offer much more than its own brand of local area network support.[18] Thus, Windows 95 includes software that ensures its host system will be fully equipped as a NetWare client machine.

Beyond its support of local area network facilities, Windows 95 has many other features that involve communications. From simple telephone line dial-up facilities to support for the latest generation of

18. Whether peer networking will literally be given away in the Windows 95 box is a packaging issue that probably won't be decided until shortly before Windows 95 ships. It may be packaged as a separately priced add-on.

mobile, handheld devices, Windows 95 aims to be about as good a client machine operating system as it can be, including

- Client support for all popular networks: Novell's, Banyan's, Microsoft's, and others.
- Multiple client support, allowing a client machine to connect simultaneously to different networks—perhaps to a Novell local area network and to a TCP/IP-based wide area network.
- A peer server capability that matches the original capability provided by the Windows for Workgroups product. Workgroups or smaller businesses can thus avoid the need to dedicate a machine to server functions.
- Electronic mail support based on the message application programming interface (MAPI) and extending to facsimile devices as well as popular electronic mail networks.
- Remote connectivity and administration features that provide efficient access to and management of a local area network over a low-bandwidth connection. Windows 95 acknowledges the "traveling PC" phenomenon in its support for file synchronization capabilities and effective data transfer over a low-speed connection. Thus, you can dial back to home base and download a copy of a document at a decent speed. When you revise the document and take it back to the office, Windows 95 helps you figure out how to synchronize your hotel room edits with the local master copy.
- Pen support. The pen-based computer revolution was predicted, and then it never really happened. Even so, there is a steady growth in the use of pen computing devices. Windows 95 incorporates support for pen-based machines. As and when the revolution occurs, your Windows 95 software will be ready.

Bringing Windows 95 to Market

Describing what the Windows 95 development team set out to accomplish begs the question of whether the product will be successful. The mission of making a Microsoft product a success involves many other Microsoft groups. Some of these groups, such as the product support division, aren't fully engaged in seeing to the success of the product

until it ships to customers. Everyone involved faces a considerable challenge. Success for Windows 95 means selling tens of millions of copies. Sales of only a few million copies (usually an indication of a runaway software bestseller) will be a commercial disaster.

Outside Microsoft, the most important group influencing the success of Windows 95 will be the independent software vendors (ISVs) courted by the company's developer relations group (DRG). If the ISVs devote their resources to writing applications for Windows 95, competing operating systems such as IBM OS/2 and Novell NetWare will suffer by comparison. Windows 95 presents an unusual selling job for Microsoft in that they must persuade the application developers to take presumably perfectly fine Windows applications and modify them. The DRG spent much of 1993 evangelizing for Microsoft's OLE technology and the 32-bit API of Windows NT that would appear in Windows 95 in 1994. Whether the benefits of OLE and the 32-bit capabilities of these operating systems are compelling enough to warrant major investment by the ISVs remains to be seen.

Microsoft provided the ISVs with a lot of early information about Windows 95 in a series of design reviews held in Redmond during the summer and fall of 1993. The audience for these events was usually fairly small (the largest made up of perhaps 100 people), and Microsoft always prefaced such an event with a warning that many product features were expected to change. The participants also had an opportunity to influence the Windows 95 design team. The team often asked for comments on possible solutions to issues that had not been entirely decided. Early on, the possibilities for change were quite numerous, but as the planned shipment date drew closer, these opportunities to influence the Windows 95 team naturally diminished.

As Windows 95 gathered marketing momentum, the product team's goals were translated into the market message behind the product. Customers are most influenced by the perceived benefits of any product, and Microsoft used the Windows 95 project goals as the basis for their initial customer presentations. In the early fall of 1993, Microsoft's first closed door product briefings identified three main benefits of Windows 95:

- Easy to use—based on the Plug and Play capability, the new shell, and the extensive use of Microsoft's OLE 2 technology.
- Powerful 32-bit multitasking system—based on the new operating system kernel, the new filesystem, and the improvements in device support.

- Great connectivity—based on the new networking components and the mobile computing enhancements.

The first of the more public product briefings was given to a group of industry journalists on May 12 and 13, 1994, in Redmond. The press rollout was scheduled to take place shortly before the Beta-1 release, which was actually supposed to be ready to hand out at the briefing and to coincide with the launch of the marketing campaign that precedes every Microsoft operating system product release.

At that rollout, the product goals were restated in short form—"easy," "more powerful," and "more connected." The marketing message has retained a degree of consistency throughout the project.

Whether these benefits are enough to sell Windows 95 to the end user is a subject for the future and for a different forum. Certainly Microsoft has every chance of success with the product. Their early 1994 estimates indicated that about 50 million copies of Windows would be in use by mid-1994, with perhaps 60 to 70 percent of all new machines shipping with Windows already installed. The principal target market for upgrading existing Windows 3.1 users will be about 60 percent of the installed base.[19]

For Microsoft—The Bottom Line

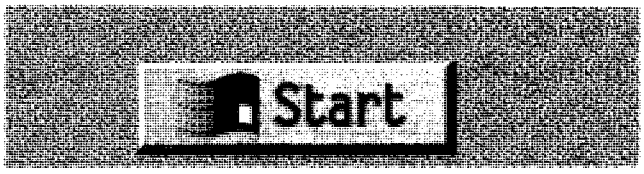
Altruism is rarely a consideration in Microsoft's business thinking. Yes, some product characteristics, such as compatibility and ease of use, are deeply ingrained in the thinking of every person in the product development groups. The Windows 95 team tried as hard as anyone to meet the ease-of-use goal, and indeed, their motivation did extend far beyond the simple desire for commercial success. However, the team also wanted to sell one heck of a lot of software. Work out the numbers and you'll see that selling a Windows 95 upgrade to every existing Windows user would translate into a billion dollars of revenue. The team knew that if Windows 95 really could achieve the "make it easy" goal, the door to more new users and more software sales would be unlocked. Building a great product was definitely the number one goal. Selling lots of copies came in a close second.

19. Microsoft classifies these users as "active" users; that is, they are people who periodically upgrade some part of their computer systems, be it hardware or software. The rest simply don't upgrade anything (and probably drive a 10-year-old car quite happily as well).

Conclusion

In this chapter, we've looked at the underlying goals and philosophy behind the Windows 95 development project and at a synopsis of the major new features. Entering a mature market, the product has to meet some stringent compatibility and performance goals as well as introduce new features that will motivate Windows users to upgrade and will attract new users to the Windows platform. Windows 95 is also an important component in Microsoft's systems software plans. Married to the strengths of Windows NT, it becomes part of an enterprise-wide computing system and introduces some of the Cairo product concepts for the first time. As our review of the development team's self-imposed ten commandments suggests, Windows 95 is also an ambitious project. How Microsoft plans to meet the target it has set for itself is what most of the rest of this book is about.

Windows 95 is an Intel processor–based operating system. The Intel family of processors has had a significant influence on both MS-DOS and Windows over their lifetimes. In return, Windows has influenced Intel's processor designs. In the next chapter, we'll look at the Intel processors and highlight the features that have an impact on the design and operation of Windows itself.



CHAPTER TWO

INTEL PROCESSOR ARCHITECTURE

Inside every fine operating system beats the heart of a good processor. In our case, it's very definitely Intel inside. Windows 95 has been designed and developed for Intel processor–based systems only. Microsoft's high-end operating system, Windows NT, broke with the Intel tradition in order to allow vendors to choose from a variety of processor types as the base for a system, and Microsoft and its development partners have introduced versions of Windows NT for the MIPS R4000, the DEC Alpha, the PowerPC, and other advanced processors. None of these chips is compatible with the Intel processor family, so the only way to get existing applications for Windows or MS-DOS to run on one of these processors is to include some form of Intel processor emulation with the Windows NT version for the processor. For a Windows NT user, the performance overhead of the emulator isn't a real problem. After all, that user bought Windows NT principally to use on a network server or to run a new native 32-bit application. Any slowdown in such a user's occasional use of an existing 16-bit Windows application isn't really an issue. There are also some thorny problems associated with running MS-DOS applications on Windows NT. The preservation of the Windows NT security model prevents a lot of older MS-DOS applications from running, for example. But running MS-DOS programs just isn't the role a Windows NT machine is meant to fill, so Microsoft decided that putting restrictions on Windows NT's 16-bit application environment was acceptable.

For a Windows 95 user, Microsoft felt that any similar restrictions or performance overhead for running 16-bit applications would be

completely unacceptable. After all, most Windows 95 users would already be using Windows on their desktop or laptop machines. Their main initial reason for installing Windows 95 would probably be to have their existing applications run faster or better. Any compatibility or performance problems for 16-bit applications would be a major barrier to the mass acceptance of Windows 95.

Thus, the Windows 95 team had to provide 100 percent compatibility and zero performance overhead to the Windows 3.1 user. Tough goals. Fortunately, Microsoft's experience with early versions of Windows, OS/2, and Windows NT had equipped them with the expertise they needed to meet these goals. Microsoft's experience also told them that the compatibility and performance goals could not be met for Windows 95 running on a non-Intel processor. Any dreams of a portable version of Windows were laid aside early on. Windows 95, and any direct successors, will forever run on Intel processor systems only.

Intel Inside

One could write a book devoted to the low-level details of Windows 95 and its interaction with the Intel processor and the system that contains it, but that is not the purpose of this chapter. We'll look at some aspects of the hardware that have to be understood in order to make sense of some of the Windows 95 features we'll look at in detail in later chapters—particularly Windows 95 memory management, its support for MS-DOS applications, and the new Plug and Play services. However, this chapter is certainly not intended to be an exhaustive treatment of the subject.[1] Most of the information in this chapter will relate to the 80386, 80486, and Pentium processors that Windows 95 runs on. A lot of the less relevant details have been left out or simplified. You may already know more about the Intel processor family than you care to remember. If you do, I suggest that you go straight to the next chapter. If you don't care to know a lot about the Intel processor family, don't worry: the rest of the chapter deals only with the details of the hardware you need to know about. We'll get back to the Windows 95 software very soon.

1. Of the many books that do provide an exhaustive treatment of hardware issues, a good one is Ross Nelson's *Microsoft's 80386/80486 Programming Guide* (Microsoft Press, 1991).

Here's what we'll look at in this chapter:

- The Intel processor family—the continuing influence of the original 16-bit Intel processor, the 8086, on all versions of Windows because of the MS-DOS software compatibility requirement
- Processor architecture and modes—the basic design of the Intel chip family and how the processor can be made to run the different application types (MS-DOS, 16-bit Windows, 32-bit Windows)
- Memory management—the different methods for handling memory allocation on the Intel 80386 processor
- Protection—how the 80386 processor allows the operating system to protect itself and to protect applications and devices from one another

The Intel Processor Family

Intel introduced its first 16-bit microprocessor, the 8086, in 1978. IBM ensured the role of Intel processors in subsequent computing history by adopting the Intel 8088 (a slightly slower version of the 8086) for the IBM Personal Computer in 1981. Microsoft (figuratively, at least) took its place on the podium with MS-DOS, the operating system it implemented for the IBM PC. Successive models of the PC, from IBM and its competitors, have continued to use Intel processor chips and copies of MS-DOS in vast numbers. Somewhere, someone is buying a PC right now. It probably has an Intel processor inside, and it probably comes with a copy of MS-DOS. This buying process is repeated tens of millions of times a year, and many fortunes, Intel's and Microsoft's included, have been made as a result.

From the software point of view, the Intel processor family has gone through two major architectural changes since 1978. These changes appeared with the 80286 and 80386 processors. From the hardware designer's point of view, there have been other major design changes, such as the integration of the processor and floating point processor capabilities on the single 80486 chip. These hardware changes, together with many other feature and performance improvements, are often denoted by product name suffixes such as SX and SL. Each change almost always meant more speed and rarely required any major

modification on the part of the operating system software designer. That was not true in the case of the major architectural revisions introduced with the 80286 and 80386 processors. At the risk of offending some hardware designers, we'll look primarily at the processor design revisions that enabled significant new software capabilities.

Backward Compatibility

The single most important aspect of the Intel processor design has been the backward software compatibility of the different chips. And successive versions of MS-DOS have ensured that this compatibility feature has been readily available to both programmers and users. Every MS-DOS program ever written for an Intel 8086 will run unchanged on a Pentium processor. This compatibility has allowed users to buy newer and better hardware with every change in processor generation and carry with them the applications they know and use every day. I'd be willing to bet that many copies of version 1.0 of Lotus 1-2-3 are still in use. Amazingly, the very first release of Microsoft Windows (1985) would actually run on a floppy disk–based PC with an 8088 processor (1981). That same software will still run on a Pentium-based system today.

Software compatibility has been the key to the success of the Intel processor family and, to a large extent, the key to the success of the whole personal computer industry. When Intel released the 80286 processor in 1982, the announcement lauded, in addition to compatibility, its higher speed and new "protected mode." Unfortunately, the protected mode wasn't compatible with the 8086. In 1984, IBM introduced its first 286-based system, the IBM PC AT. Microsoft didn't try to exploit the protected mode with the MS-DOS release (version 3.0) for the PC AT. MS-DOS used the 286 simply as a faster 8086. However, Microsoft did release XENIX, its UNIX-derivative operating system, for the PC AT. XENIX was the first operating system that tried to exploit the 286's protected mode of operation. But XENIX didn't try to provide MS-DOS software compatibility. A few years later, the designers of OS/2 made valiant attempts to exploit the 286's protected mode while retaining that all-important property, MS-DOS software compatibility. There were many shortcomings.

If all of this sounds confused, it was. In truth, Intel's implementation of 8086 compatibility alongside the 286 protected mode feature was poorly designed. For example, once an operating system had switched the processor into protected mode operation, there was no way of switching back to real mode other than by simulating a complete

reboot of the machine! This and other deficiencies meant that the 286 processor was rarely used as anything other than a faster 8086. However, the mistakes with the 286 design and the early experience from operating system projects such as OS/2 ensured that the next processor in the family—the 80386—came out right. The 386 offered 8086 compatibility, 286 compatibility (which ultimately might not have been worth the microcode), a new 32-bit mode (*386 native mode*), and an unusual new mode of operation called *virtual 8086 mode*. This last feature enabled the implementation of an operating system that could run not just one, but many MS-DOS programs compatibly *and* simultaneously. Microsoft helped Intel design virtual 8086 mode and harnessed that mode initially with the release of Windows/386 in 1987. Other operating systems—Quarterdeck's DESQview, IBM's OS/2 version 2.0, and many versions of UNIX—also used the virtual 8086 feature to good effect. The successor processors in the Intel family, the 80486 and the Pentium, preserved the virtual 8086 mode feature, and today most operating systems, including Windows 95, continue to exploit it.

The most recent releases of Windows have been designed only for the 80386, the 80486, and recently, the Pentium processors. Essentially, Windows has treated each of these processor types as a 386. A number of low-level processor features have to be managed differently, but none of this low-level management is visible to an application program or indeed to most of the Windows operating system itself. Thus, we won't get into the intricate details of, for example, how Windows 95 manages floating point operations on the different processor types. In the rest of this book, you'll see references to only the 386 processor. Read this to mean "386, 486, or Pentium." The keys to understanding how Windows exploits the Intel 386 processor architecture are in its management of memory, its processor modes, and its protection scheme. That's what we'll look at next.

Processor Architecture

The Intel 8086 introduced a microprocessor memory architecture referred to as *segmented addressing*. Similar schemes had appeared in the design of other, generally much larger, computers, but the 8086 was the first major microprocessor to employ the technique. Since all MS-DOS programs throughout the 1980s were written for compatibility with the 8086 (and Windows 95 still has to be able to run those programs), it's important to understand the 8086 memory architecture.

The 8080 and 8086 Processors

The 8-bit predecessor of the 8086—the Intel 8080—allowed a program to address a total of 64 kilobytes. Each addressing register of the 8080 was 16 bits. Sixteen bits gave you 65,536 total addresses and thus 64K of address space. Intel tried pretty hard to make the 8086 compatible with the 8080 and did preserve the 16-bit address registers. Intel's goals for the 8086 were much loftier, however, and they added four *segment registers* to the 8086, allowing a program to address up to 1 megabyte of memory. Essentially, a segment register points directly to the first byte of a memory segment. A segment can begin at any 16-byte chunk of memory (what Intel called a *paragraph*). Adding 1 to a segment address points you to a memory address 16 bytes higher in memory. Using this segment address as a base address (that is, as address zero for this segment), the programmer can then use another processor register to reference any byte within the subsequent 64K. The processor simply combines the contents of the segment register and an address register to form a unique 20-bit address. Twenty bits gives you 1,048,576 total addresses and thus 1 MB of address space. Figure 2-1 shows how the 8086 performs the address arithmetic. Note that the operation of combining the contents of the segment register and the address register to obtain the final memory address is carried out by the processor itself. No direct action is required on the part of the programmer.

The segment registers on the 8086 have to be manipulated by the programmer. When the operating system loads an application, it initializes the segment registers before running the application. After that, the application code manipulates the segment registers as it needs to. Most early MS-DOS programmers and compiler writers learned many tricks for efficiently using the 8086 segment registers.

This segmented memory architecture has been both a boon and a pain for software writers. On the plus side, the segmentation allowed the use of techniques such as expanded memory—with a combination of software and hardware tricks, segments of 8086 memory could be temporarily replaced, effectively increasing the total memory available to a program. On the minus side, segment management was a chore for anyone developing large (that is, larger than two 64K segments) applications.[2] Scanning through a 100,000-element array of 2-byte integers,

2. During the development of the first version of Windows, signs proclaiming *SS != DS* were popular in many programmers' offices. The signs were intended to be a constant reminder to the developers. They hoped the signs would lead to fewer bugs.

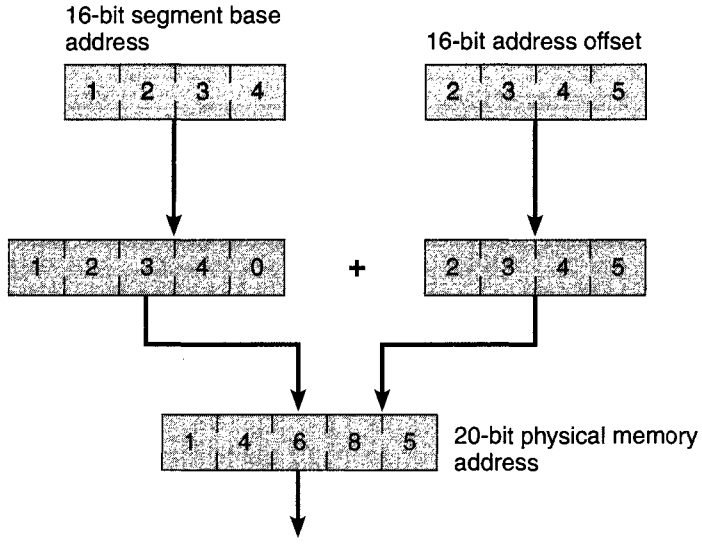


Figure 2-1.
Intel 8086 address calculation.

for example, meant reloading the appropriate segment register at least three times during the scan. Programmers used to larger machines, or to microprocessors such as Motorola's 68000, were more accustomed to a *linear address* scheme. With a linear addressing architecture, the programmer would simply increment a single (usually 24- or 32-bit) address in order to scan the entire physical memory present on the system.

The 640K Barrier

The 1-megabyte memory limit of the 8086 architecture never received wide public attention. Instead, the infamous 640K limit in DOS was the popular target for much ire and ill-informed criticism. So where did the 640K limit come from? The designers of the original IBM PC decided to reserve 384K of the 8086's enormous 1-megabyte address space (remember, this was 1981) for hardware and system software purposes. The remaining 640K was free for use by DOS and application programs. Within the upper 384K were the BIOS code, screen memory, and other system elements. Figure 2-2 on the next page is a reproduction of the first published memory layout of the original IBM PC.[3]

3. *IBM Technical Reference #6025005*. The first edition was published in August 1981.

System Memory Map

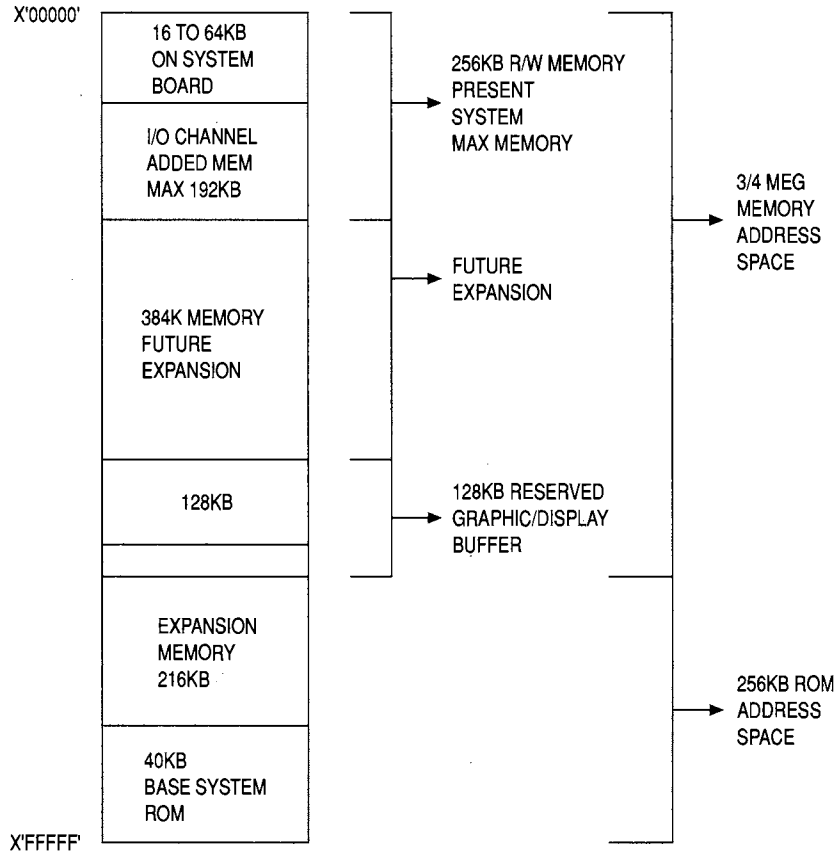


Figure 11. SYSTEM MEMORY MAP

Figure 2-2.
The first published memory map for the original IBM PC.

DOS really had little part in determining the 640K limit, and the layout for the first megabyte of memory on a PC still has an impact on the design of operating systems today. If you want to build an operating system that runs MS-DOS programs, many of which expect to find certain resources at the specific addresses chosen in 1981, you have to develop some method for supporting this memory layout.

The 80286 Processor

Enter the 80286 and protected mode operation. Once again, software compatibility was a key goal in the design of the processor, so the 286 designers retained the basic instruction set and addressing method of the 8086. Indeed, at power on, the 286 operates in *real mode* (a term coined at that time to designate operation in 8086 mode) and behaves for all intents and purposes just as an 8086 does. But Intel added the new *protected mode* of operation to significantly increase the processor's capabilities. An operating system can programmatically switch the 286 from real to protected mode, and in protected mode, the processor's segment registers are used very differently.

In protected mode, the processor uses the contents of a segment register to access an 8-byte area of memory called a *descriptor.* Within the descriptor is the information that determines the actual physical address of the memory location the program is trying to reference. Figure 2-3 on the next page shows how the 286 combines the segment register, descriptor information, and address register to produce a 24-bit physical memory address. It's like having a key to a numbered safety deposit box that contains the real address of the location for a rendezvous. The segment register actually contains an index into a table of descriptors. Each descriptor can be set up to address a different area of physical memory. (Note that in descriptions of protected mode operations, the term *selector* is customary for describing the contents of the segment register. Since the value in the register isn't actually a memory address, there is some justification for yet another term.)

A descriptor contains a lot more information, related primarily to memory protection issues. The operating system sets up all the descriptors for a particular program within a contiguous area of memory called a *local descriptor table,* or *LDT.* Each program running on the 286 has its own LDT. The operating system also sets up a *global descriptor table,* or *GDT.* The operating system uses the GDT to allocate memory for itself and, for example, to allow several programs to access the same area of physical memory. The operating system can place the GDT and each application's LDT anywhere in memory. Two special hardware registers, the GDTR and the LDTR, are set up to contain the base addresses of the tables for the currently executing program. When the operating system switches tasks, it will typically change the base address in the LDTR. Usually, the GDTR remains unchanged while the system is running. Reloading the GDTR and LDTR registers is a privileged operation performed only by the operating system. The system does not allow application programs to modify the contents of these registers.

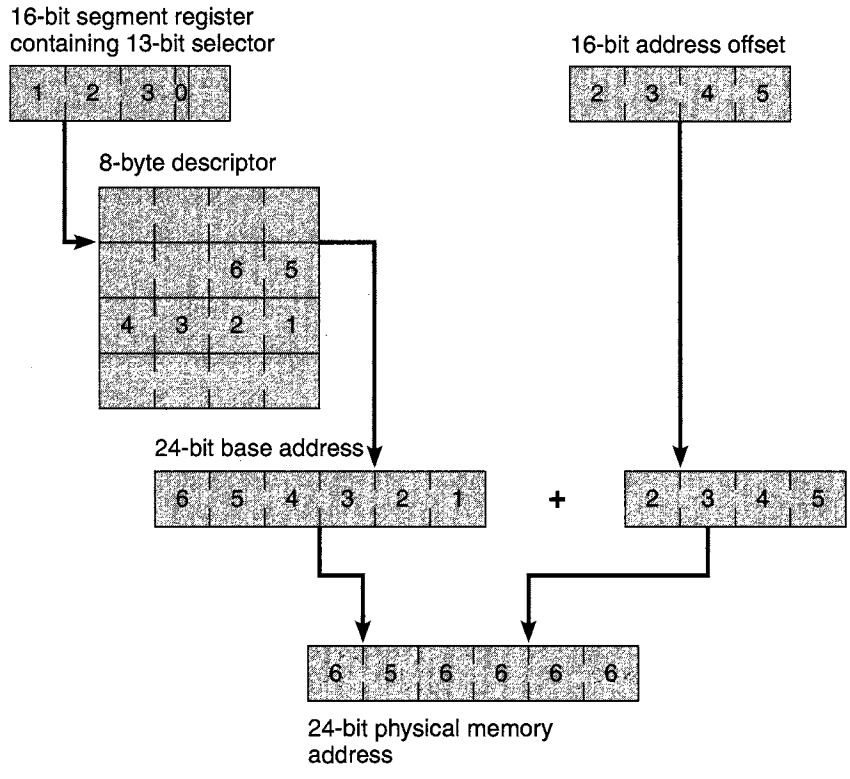


Figure 2-3.
Memory access on the 80286 processor in protected mode.

Two aspects of the new protected mode architecture are important to note.

- Protected mode introduced the notion of memory protection. Unless a program's LDT contains a descriptor for a particular area of memory, there is no way for the program to access that part of memory. Thus, an operating system can set up an environment in which several programs run concurrently, each in its own protected memory area. The 286 actually has protection capabilities beyond this, and we'll look at all the details when we examine the 80386 processor. Typically, the OS uses the GDT descriptors to allow different programs to access the same area of physical memory.
- The architecture's provision for indirect access to memory via the LDT or GDT allows the operating system to use any

suitable area of physical memory as a segment. The segments of one program need not be contiguous and can even be different sizes. As far as the program is concerned, it has access to all the memory described by its LDT. The program doesn't know, or care, exactly where in physical memory the segments exist. Figure 2-4 shows how such an allocation of memory might appear within a system running two programs that share access to one particular memory segment.

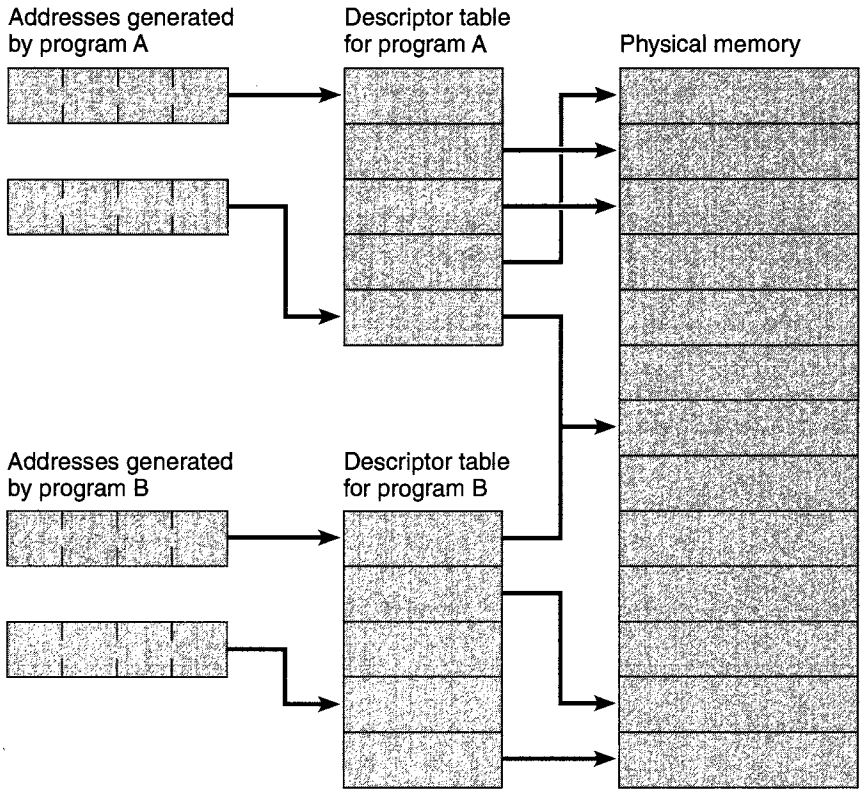


Figure 2-4.
Hypothetical memory allocation for two programs running on an 80286 processor in protected mode.

The 80386 Processor

Note that the 80286 retained the 8086's awkward segmented addressing scheme. A programmer, or a compiler and linker, still had to be

sure to set up segment registers with the correct selector, and the code that could scan through that ubiquitous 100,000-integer array still was not pretty.[4] This deficiency alone made Motorola's 32-bit microprocessor family the almost unanimous choice for manufacturers designing UNIX workstations. Intel had to respond to this market pressure, and they did, introducing the 32-bit 80386 processor in 1987.

Microsoft worked closely with Intel during the 80386 design phase and strongly influenced the capabilities of the new *virtual 8086 mode* supported by the 386.[5] Microsoft's interest in the project was to make sure that the 386 included all the capabilities necessary to allow new operating systems to run existing MS-DOS programs. Microsoft had a lot of battlefield experience from meeting this requirement over the course of several operating systems and versions of operating systems, and the work they'd put into OS/2, MS-DOS 95, and Windows, all for the 286 processor, had persuaded them that there had to be an easier way. Sometimes silicon chips don't turn out quite the way the designers intended, but in the case of the 80386, Intel got it right. The new 32-bit capabilities and the virtual 8086 mode feature worked well from the time of the first production samples of the 386, and apart from changes to internal details, those features remain the same in the 80486 and Pentium processors.

Windows 95 is a 386 operating system, so we need to take a close look at the features of the 386 (and by extension of the 486 and the Pentium) that are important to Windows 95's operation. Software compatibility for the now enormous installed base of MS-DOS software remained an overriding consideration, so PC manufacturers[6] first released systems that used the 386 as a yet faster 8086—turn on the power and the 386 runs in real mode, precisely emulating the 8086. However, the 386 evolved from the 286 in a number of distinct ways, all of which called for a new operating system to make the new features of the 386 available to application programs:

- Internally, everything grew from 16 bits to 32 bits—all the registers, the memory addresses, and so on.

4. If you're interested in the more amusing aspects of microprocessor history, you might like to revisit Intel's 286 sales campaign of the time. Their explanation of why a segmented architecture beats a linear architecture is a triumph of marketing over science.

5. In fact, the I/O permission bitmap, so important to virtual mode operation, was present in the 386 largely because of Microsoft's lobbying.

6. Compaq was the first company to introduce a PC that used a 386 processor, and this was the first time that one of the so-called "clone" manufacturers broke ranks. Compaq's low-risk bet helped push IBM out of its industry leadership position.

- Although the 386 preserved the notion of segments, a single segment could now be 4 gigabytes in size as opposed to a mere 1 megabyte. For all intents and purposes, the programmer could now treat the 386 as though it had a linear address space. Intel finally had a real 32-bit microprocessor.
- The 386 improved the memory protection scheme further. An operating system designer could now implement a full *virtual memory* scheme on the 386. (Note that virtual memory and virtual 8086 mode really aren't related, terminology notwithstanding.)
- An operating system could switch the 386 processor at will among its different operating modes. The properly equipped 386 system could run 8086, 286, and new 32-bit 386 programs simultaneously.
- The virtual 8086 mode and the associated *I/O permission bitmap* allowed the implementation of complete MS-DOS software compatibility within a protected multitasking system.

80386 Memory Addressing

The 80386's software compatibility features ensure that in real mode it operates just as an 8086 does. Address construction is the same as for the 8086, and all extraneous information (notably the high-order 16 bits of each register) is simply ignored during execution. In protected mode, the operating system that controls program loading and execution must set up a program's descriptor table in such a way that the processor knows how to interpret the memory address information. The protected mode process for calculating a physical address on the 386 is similar to that of the 286: the processor uses the contents of a segment register as an index into a descriptor table, and the descriptor table entry contains nearly all the remaining necessary information—"nearly" all because the 386 allows an operating system to implement a complete paged virtual memory scheme. When the operating system enables paging, the address information extracted from the descriptor table must go through a further level of interpretation before it is used as an actual memory address.

80386 Descriptor Format

Figure 2-5 on the next page illustrates the layout of a single descriptor table entry on the 386. Let's look at each field in a little more detail.

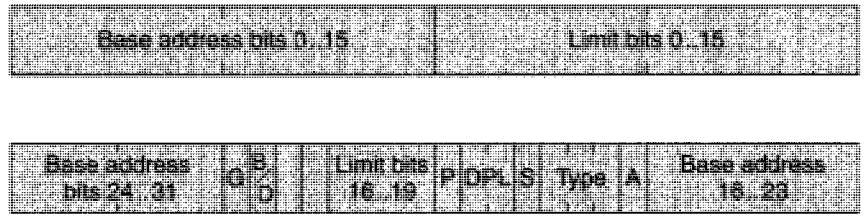


Figure 2-5.
80386 descriptor table entry format.

Base Address The processor forms a 32-bit address from the four base address fields. Once assembled, the address specifies the first memory location of the memory segment the program wants to reference. Adding the 32-bit offset address generated by the program completes the address of the memory reference. For a 286 program, byte 7 of the descriptor (bits 24 through 31 of the base address) is always 0, since the 286 can deal only with 24-bit base addresses.

This arrangement is the basis of the addressing mechanism for 32-bit programs. Each program has to deal only with a consistent 32-bit linear address. The operating system sets up the base register to point to the first byte of the program's code or data segment, and no further segment manipulation is necessary. Since a 32-bit quantity provides such an enormous address space, only a tiny number of programs will ever need to indulge in segment register trickery.

This absence of the need for segment register manipulation is an important performance benefit. On the 286 running in protected mode, every time the contents of a segment register change, the processor must check to see that the new selector is a valid one—that is, that the new segment register contents address a memory segment allocated to the program. If the selector is not valid, the processor generates a *general protection,* or *GP, fault.* This selector validation process consumes many processor cycles, and when segment registers are frequently changed, as they must be on the 286 running in protected mode, overall program performance degrades. On the 386, most programs will never reload the segment registers and consequently never suffer the performance hit.

Limit Two fields form the 20-bit *limit* quantity, which specifies the upper limit of the memory segment addressed by the descriptor. Twenty bits, as a byte address, is only 1 megabyte. But didn't we just say that segments could be 4 GB in size, rather than just 1 MB? Read on.

G Bit The single *granularity* bit specifies whether the processor interprets the limit field value as *byte granular* or *page granular*. Byte granularity means that the processor interprets the limit value in terms of bytes. This setting (0) assists in running 286 programs correctly. Page granularity means that the processor interprets the limit value in terms of pages. Memory pages on the 386 are 4K in size, and 20 bits' worth of 4K pages equals, lo and behold, 4 GB of memory.

D or B Bit This bit is the D bit if the memory segment contains program code. The value 1 means that the segment contains native, that is, 386, instructions. The value 0 means that the segment contains 286 code. This bit is the B bit if the segment contains data. In this case, the value 1 means that the segment is larger than 64K.

P Bit The *present* bit denotes whether the memory segment is present in physical memory. This information is an important aspect of the virtual memory scheme implemented by Windows 95 since it allows the operating system to differentiate between an invalid memory reference—one in which the program tries to access memory it doesn't own—and a reference to a memory segment that has been temporarily swapped out to the hard disk.

DPL The 2-bit *descriptor privilege level* field specifies the privilege level for the segment—zero through three. The contents of the DPL field, together with the privilege level of the currently running program, play an important role in the Windows 95 protection system. Code running at ring zero, as the terminology goes, has the privilege of executing certain instructions that ring three code does not. Code at ring three, for example, can't turn interrupts on and off. Windows uses only two privilege levels—zero and three—despite the fact that the processor also supports privilege levels one and two. Someday there may be a good reason to use the extra privilege levels, but it hasn't come along yet.

S Bit The *segment* bit is always set to 1 for a memory segment. The value 0 means that the descriptor references something other than memory. The "something other" can be one of several special data structures used by a 386 operating system to control aspects of device interrupt handling and memory protection.

Type Field The 3-bit *type* field specifies the memory segment type—for example, an execute-only code segment or a read-only data segment. The

contents of the type field help the operating system maintain memory protection. An attempt to modify the contents of a read-only data segment would obviously be an error, for example.

A Bit The *accessed* bit indicates whether any program has referenced the memory segment. Any reference to the segment causes the accessed bit to be set to 1. The Windows 95 memory manager uses the accessed bit in its virtual memory scheme. If a memory segment has never been accessed while in physical memory, the physical memory it occupies becomes an excellent candidate for the operating system to reclaim it and allocate it to another program when the need comes up. And if there has been no access to the segment, it obviously has never been modified, so Windows can reclaim the memory for another use without having to write the segment out to disk.

The Descriptor in Summary

As you can see, the layout of a 386 memory descriptor is hardly the most elegant data structure ever devised. The layout is really an artifact of the earlier processors with which the 386 has to remain compatible. However, the descriptor does contain the information necessary to implement a fully protected multitasking system with virtual memory support. Windows 95 implements exactly that, and apart from the first hardware initialization sequence after power on, Windows 95 always runs in 32-bit protected mode with virtual memory enabled.

Virtual Memory

Simply put, virtual memory is a method for allowing several concurrently running programs to share the physical memory of the computer. (Note again that *virtual memory* and *virtual mode,* or *virtual 8086 mode,* are very different. The phrase *virtual mode* refers to the operation of the 386 processor in virtual 8086 mode. The context will determine the meaning of any other use of the word *virtual.*) The techniques for implementing and managing virtual memory date from many years before the introduction of the 386.[7] In fact, the early research on virtual memory was so good that the most effective techniques for handling virtual memory have changed very little since its earliest implementations. The

7. Over the years, many manufacturers and research institutes have laid claim to the "first" distinction. The earliest implementation of virtual memory was probably the one by the Atlas research group at the University of Manchester, England, during the late 1950s and early 1960s.

management of virtual memory is entirely under the control of the operating system. As far as any individual program is aware, it has access to all the memory it needs all the time. A simple example should illustrate how Windows 95 manages virtual memory.

Let's say that we have a Windows 95 system with 4 MB of memory and a hard disk with plenty of free space. Windows 95 itself, with the Shell, the Print Manager, and so on, might take up a megabyte of the available memory. On the disk is a word processing program we decide to run. Once loaded, this program occupies 2 megabytes, and we load in a large document that includes several different fonts. Altogether, this document consumes 400K of the remaining megabyte of memory. Now we decide that we need to incorporate a table of numbers in the document. The numbers reside in a spreadsheet, so we have to run the spreadsheet application to cut and paste a copy into our document. Windows 95 obligingly loads the spreadsheet application and its data into the remaining 624K of memory. Well, maybe—if we still used VisiCalc it could. Obviously, this software and data won't all fit into memory at the same time. But from our user point of view, things do work exactly as described. The system and both applications are running, so to us it seems that everything must be in memory. Everything is actually held, not in the available 4 MB of physical memory, but in virtual memory.

Virtual Memory Management

The system's virtual memory is made up of the RAM in the computer and the Windows swap file on the hard disk. The operating system manages this total available memory by swapping program and data segments back and forth between RAM and the swap file. For example, if the instructions in a particular code segment are to be executed, the segment must be loaded into RAM. Other code segments can stay on disk in the swap file until they're needed. A disk data buffer area within a data segment has to be in RAM if the disk transfer is to succeed. Whenever a segment is not held in RAM, the operating system can mark its absence by clearing the present bit in the appropriate segment descriptor. Then, if an access to that segment is attempted, the 386 will generate a *not present interrupt* that notifies the operating system of the problem. The system will arrange to load the missing segment into an available area of RAM and then restart the program that caused the interrupt. All of this swapping and notification is transparent to the application program. It's up to the operating system to carry out these housekeeping activities.

Good Virtual Memory Management

Of course, the art of designing a good virtual memory system revolves around issues such as how much of a program to keep in RAM at any one time and which segments to move from RAM to disk when RAM is full and the system needs space for a new segment. A poor virtual memory manager can slow the system down considerably. Since copying from the disk and copying to the disk are relatively slow operations, the goal of good virtual memory management is to minimize the total number of swap operations. After all, if the operating system is busy swapping, programs aren't running and no useful work is getting done.

The 386 helps things a lot by allowing the implementation of a *paged* virtual memory scheme that allows the operating system to carry out all memory allocation, de-allocation, and swapping operations in units of pages. On the 386, a memory page is 4K and each memory segment is made up of one or more 4K pages. (Small page sizes are generally more efficient because many programs exhibit a trait called *locality of reference.* For example, a program might repeatedly execute only a few instructions to scan through a text file searching for a particular string of characters. Allocating a single page for the program's code and a single page for a data buffer could satisfy this program's memory requirements for several seconds, even though the program is, in total, much larger.) Windows 95 implements such a paged virtual memory system. You'll often run across the words *paging, page file,* and *page fault* in descriptions of memory management operations. These terms are essentially identical to the *swapping, swap file,* and *not present interrupt* terms used in the earlier description of virtual memory management.

As you can see if you study the 386 segment descriptor format in Figure 2-5, there appears to be no way to allocate memory in units as small as a 4K page without wasting a lot of the memory. The trick is in the interpretation of the address once the operating system enables paging. During initialization, the operating system will first switch the processor into protected mode and then enable paging operation. Once enabled, paging stays on until the system shuts down. With paging enabled, the 386 alters the interpretation of the 32-bit address first obtained by adding the base address from the descriptor to the offset generated by the program. Figure 2-6 illustrates the splitting of this 32-bit quantity into three parts. The top 10 bits (31 .. 22) are an index into a page table directory. Part of each 32-bit quantity in a page table directory points to a page table. The next 10 bits of the original address (21 .. 12) are an index into the particular page table. Part of each page table

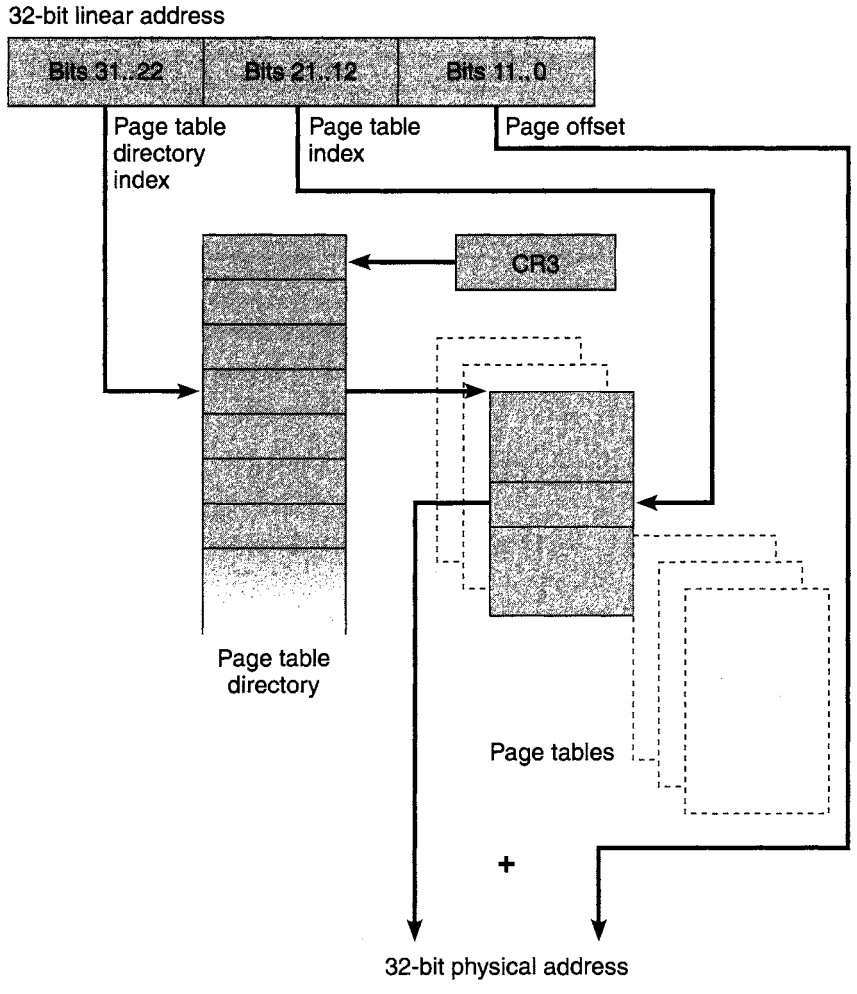


Figure 2-6.
80386 paged virtual memory address decoding.

entry points (finally) to a page of physical memory, and the remaining 12 bits of the original address (11 .. 0) make up an offset within this page of memory. The operating system anchors the entire structure by storing the address (for once, a physical address) of the page table directory for the current program in a special processor register called *CR3*. Each time the operating system switches tasks, it can reload *CR3* to point to the page directory for the new program. Although it sounds laborious, the whole address decoding process takes place at lightning speed within the chip itself. Memory caching techniques ensure that

frequently used page table entries are available with no additional memory references.[8]

To fully support the virtual memory scheme, page table entries contain more than just the address of where to find the next link in the chain. Figure 2-7 shows the contents of a single 32-bit word in both the page table directory and page table entry structures. The page table directory and each page table consume one 4K memory page (1024 entries in each). If you care to do the math, you'll see that this allows the entire 4 GB of a program's address space to be properly addressed. However, look at the numbers: a page table directory that points to 1024 page tables could mean that the system has to use 4 MB of memory (1024 page tables, each 4K in size) simply to store the page tables. Fortunately, the flag bits in the page table directory allow the system to store the page tables themselves on disk in the paging file. Thus, if you run a very large program (for example, a 1-GB program, which will need 256 page table pages), the system will swap page tables as well as program code and data pages in and out of memory.

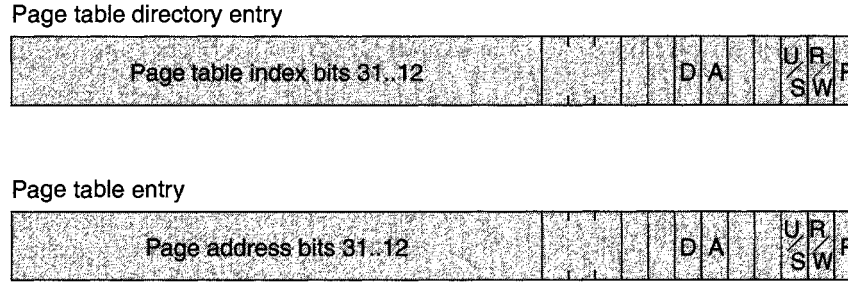


Figure 2-7.
80386 page table directory entry and page table entry formats.

To fully support the virtual memory operations and the 386 memory protection system, the page directory and page table entries include a number of flag bits. The processor itself modifies some of these flags directly. The operating system manages others. Let's look at a few of these fields in detail.

8. Intel's experiments indicate that the required page table entry is found in the cache more than 98 percent of the time.

D Bit Whenever a program modifies the contents of a memory page, the processor sets the corresponding page table *dirty* bit. This tells the operating system that if it wants to remove the page from memory to free up space, then it must first write the page out to disk to preserve the modifications.

A Bit Any reference—read, write, or execute—to a page causes the processor to set the *accessed* bit in the corresponding page table entry. The virtual memory manager can use this flag to figure out whether it's wise to remove a particular page from memory. A page with the access bit clear for the last 10 seconds, for example, has never been accessed. Removing that page from memory is probably a better choice than removing a page that was definitely in use during the same time period. Windows 95 uses a standard algorithm known as *least recently used (LRU)* to determine which page to remove from memory. The more recently used a page, the less likely it is to be re-allocated.

P Bit The *present* bit is set to 1 only when the page table or memory page addressed by the table entry is actually present in memory. If a program tries to reference a page or page table that is not present, the processor generates a not-present interrupt and the operating system must arrange to load the page into memory and restart the program that needed the page.

U/S Bit The *user/supervisor* bit is part of the 386's overall protection system. If the U/S bit is set to 0, the memory page is a supervisor page—that is, it is part of the memory of the operating system itself—and no user-level programs can access the page. Any attempted access causes an interrupt that the operating system must deal with. In Windows 95, as in earlier versions of Windows, this illegal memory reference might lead to one of the now infamous *General Protection Fault* messages. Since any such access attempt is the direct result of a bug in the application program, it's hard to know what else to do with the offending program.

R/W Bit The *read/write* bit determines whether a program that is granted access to the corresponding memory page can modify the contents of the page. A value of 1 allows page content modification. A value of 0 prevents any program from modifying the data in the page. Normally, pages containing program code are set up as read-only pages.

Mixing 286 and 386 Programs

As we have seen, the 286 and 386 processors interpret the contents of their internal registers and the resultant memory addresses in very different ways. Nearly every Windows application program to date has been written and compiled as a 16-bit program—meaning that it uses the instructions and memory addressing operations of the 286 processor. One of the major improvements in Windows 95 is its support for 32-bit programs that use the instructions and memory addressing operations of the 386 processor. Windows 95 itself is a mixture of 16-bit and 32-bit code. Mixing the two programming models efficiently is a major development challenge.

The major problem is allowing 32-bit code to make calls to 16-bit code and vice versa. Since the memory address formats are completely different—32-bit base address and 32-bit offset vs.16-bit segment register and 16-bit offset—simply jumping between 32-bit and 16-bit code is insufficient: the memory address format must also be changed.

To mediate between the two models, Microsoft developed a technique it calls *thunking*. A *thunk* is a short sequence of instructions responsible for converting the memory addresses from one format to the other. For example, when a 32-bit application makes a call to a Windows User function, the Windows kernel accepts the call and its 32-bit parameters and then calls a thunk. The thunk translates the parameters and addresses to 16-bit equivalents and then calls the 16-bit User routine.[9]

The efficient operation of the *thunk layer*, as it's called, is critical to the performance of Windows 95. In Chapter 4, we'll look at exactly how Windows 95 uses its thunk layer.

The Protection System

Any modern operating system must offer protection capabilities: protection of the user's data, protection of one program from others running concurrently in the system, and protection of physical devices from unauthorized access. Windows 95 harnesses all of the 386's protection facilities to deliver these capabilities.

9. User is one of the Windows 95 components still implemented as 16-bit code. Compatibility issues coupled with the project schedule were the principal reasons that User didn't get translated to 32-bit code.

Memory Protection

We've already seen some aspects of the 386 protection mechanism that relate specifically to memory protection:

- The provision for the operating system to set up page tables that describe exactly the areas of physical memory a program can access
- The read/write page table entry flag that prevents a program from modifying the contents of a read-only page or a program code page
- The user/supervisor flag that allows the operating system to protect all of its own memory from any access by an application

Whenever an application tries to access a memory location that is not within its current memory map, the 386 processor generates an interrupt and hands the operating system a collection of information about the problem. In a couple of cases, the memory reference will actually be quite legal and the operating system must arrange to add the appropriate memory page to the application's memory map. For example, a function call within the application can push onto the program stack parameters whose requirements exceed the memory currently allocated to the application. The operating system responds by arranging to add pages to the application's stack space and then restarts the application as if nothing had happened. With applications for Windows, there are also cases in which the operating system would like to allocate more memory to an application but has simply run out.[10] Sometimes the user sees a dialog box that says system resources are too low to continue, and sometimes the application simply fails. Windows 95 reduces the likelihood of this type of problem by greatly expanding the number of available operating system resources. Essentially all system resource requests are now satisfied by the operating system's allocating memory from a 32-bit protected mode memory pool.

In still other cases, an invalid memory reference message might indicate some sort of software problem—an application's incorrectly trying to access memory past the end of one of its data structures, for instance—and the system would have no choice but to terminate the

10. The most common case of this, under Windows 3.0 and 3.1, is exhaustion of the 64K GDI heap space.

offending program. Those of you who have used earlier versions of Windows will, no doubt, have seen enough *Unrecoverable Application Error* and *General Protection Fault* dialogs to be familiar with the handling of such a situation.[11] Fortunately, the quality of Windows development tools and application testing has now reached a level that makes this type of error rare.

Operating System Protection

There is more to protection than memory management. There has to be a way to prevent applications from maliciously or inadvertently corrupting the operation of the system. The several special 386 instructions that deal specifically with task switching, interrupt handling, and other system management issues are cases in point. Clearly, the Windows 95 kernel has to be the only software able to perform these operations. If an application could interfere with these delicate operations, mayhem would be bound to ensue. The 386 provides for this protection requirement by maintaining as many as four processor *privilege levels.*

Software running with privilege level zero can do anything it wants to: change page tables, switch processor modes, turn paging on and off, halt the processor, and so on. The Windows 95 operating system executes with privilege levels zero and three. Applications run only with privilege level three and are subject to its several restrictions. A program with privilege level three that tries to execute any of the privileged instructions—specifically the task switching, interrupt handling, and system management instructions mentioned earlier—will cause the processor to generate an interrupt. The operating system will retrieve the interrupt information and will, most likely, terminate the offending program.

The 386 has some complex mechanisms for managing software running at any of the four privilege levels. You'll hear the phrase "running at ring three," for example, meaning that the processor privilege level is set to three for the program in question. The more privileged

11. In fact, most UAEs under Windows 3.0 came from an application's making Windows function calls using incorrect parameters. By the time the system would figure this out, it would have no choice but to terminate the offending program. Windows 3.1 added parameter validation. An application's passing illegal parameters to the system resulted in an immediate return of an error to the application. Some applications couldn't handle the error return and failed in strange ways.

the software is (that is, the lower its privilege level), the more it can do to affect the operation of the system or of other programs running under the system.

There has to be some controlled way for the processor to switch between privilege levels—when an application program calls an operating system service, for example, or when a hardware interrupt causes a device driver to execute. The 386 provides for this switching by means of a *gate*, a specialized descriptor table entry that allows control transfers to occur between rings. There are actually four different types of gate: *call*, *interrupt*, *task*, and *trap*. A call to the operating system, a hardware interrupt, or an error condition such as a protection fault causes an entry to ring zero code via a gate. As processing is en route to a more privileged execution level, a new instruction pointer and stack pointer come into use and some sensitive data is stored in a protected area of memory. The corresponding return to a less privileged level restores the context of the less privileged code. Since it is the operating system that sets up the gates originally, the operating system remains in control of what happens during these transitions—ensuring that system integrity isn't compromised.

Device Protection

The device protection issue revolves around correctly sharing a resource, such as the hard disk, or preventing two programs from both trying to use a nonshareable device, such as a COM port, at the same time. Windows 95 handles a lot of the device management issues itself, but the 386 also has a significant part to play.

Low-Level Device Access

At the basic hardware level, a program controls all input/output operations by manipulating the processor's *I/O ports* and *interrupt requests* (usually referred to as *IRQs*). You've probably installed in your PC adapters whose documentation refers to their use of specific I/O addresses and IRQs. Adding a third serial port (the COM3 device) to a system usually involves much frustrating effort to prevent conflicts between the third COM port and the existing COM ports. The conflicts in question are those between the I/O addresses and the IRQ. Unless you set up the third COM device with a unique I/O address and IRQ, the controlling software can't determine which device it needs to take care of when an I/O request is made.

From the inside looking out, the I/O ports appear to be similar to a memory address. There are a total of 65,536 (64K) possible I/O ports on the 386, though the majority of them are never used. Programs control devices by reading from and writing to the appropriate I/O ports by means of special instructions. In the case of a COM device, placing a byte of data in the appropriate I/O port will cause the data to be sent down the attached wire. An interrupt manifests itself as a temporary pause in the processor's current activity, coupled with the execution of a piece of software that has been specifically set up to be responsible for dealing with the interrupt. When a hardware interrupt occurs, the 386 arranges an orderly suspension of the current program and then begins execution of some other code from within the operating system. A device generally initiates an interrupt whenever it needs attention—when a data transfer has been completed, for example. The processor and associated hardware take care of generating interrupt signals and moving bytes in and out of the I/O ports. The operating system is responsible for installing and configuring the various routines that manage the data transfer process and other housekeeping activities.

High-Level Device Access

Windows 95 and most other operating systems control peripherals by means of *device drivers*. These software modules control all aspects of a device's operation—moving data to and from memory buffers, handling interrupt requests, and so on. An application requests access to a device by making a device open call to the operating system. If the call is successful, the application can then read and write data with a further series of system calls and, finally, close the device. This holds true whether the device is a single resource such as a COM port or a shared resource such as the hard disk. In the case of the hard disk, the open request is obviously for a file on the disk rather than for the disk itself. In this ordered world, device management is relatively easy and the system concerns itself most with the efficiency of the I/O operations. All these application requests are defined as part of the Windows API. The operating system validates the API calls, hands them to the appropriate device driver, and assists in error management and task scheduling.

Unfortunately, it isn't that easy when you want to run MS-DOS applications concurrently with Windows applications. In particular, many MS-DOS applications believe that they are in total control of the system. They don't try to account for other applications that might be running simultaneously with them, and they may try to access device

hardware directly. For example, most terminal emulation programs will manipulate the COM port I/O addresses without making any operating system requests. This direct access leads to a number of problems on a Windows 95 system when you want to allow simultaneous execution of more than one MS-DOS application:

- Two applications could try to access the same device at the same time. There has to be some way to prevent this conflict.
- Typically, a 386 program that controls a device directly is running at ring zero. If Windows 95 allowed an application to do this, that application would have access to other privileged system resources. To protect other programs, such privileged execution must be avoided.
- A program that believes it is in sole control of the system might sit forever in a loop waiting for something to happen—a key depression or a character from a COM port, for example. If no other program can run at the same time, the performance of the whole system sinks to nothing. This kind of dominance has to be prevented.

Using the 80386 Device Protection Capabilities

Windows 95 uses a whole range of tricks to avoid these device access problems while still allowing older MS-DOS programs to run without modification. And the 386 provides one hardware feature crucial to the successful implementation of this MS-DOS program support: the I/O permission bitmap, a hardware mechanism that allows Windows 95 to manage device access for every program running on the system.

Whenever Windows 95 starts a new application, it determines whether the application is a Windows application or an MS-DOS application. Windows applications all use operating system APIs to access files and devices, so each Windows application runs at ring three and has no permission to access any device directly. A Windows application will request access to all devices by means of API calls. If the Windows application does try to access a device I/O port, the 386 will signal a protection fault to the operating system and Windows 95 will terminate the offending application. Each time the user starts an MS-DOS application from the Windows 95 shell, the application will be set up to run in virtual 8086 mode in a new *virtual machine (VM)*. Windows 95 must account for the possibility that the MS-DOS application might try to

directly access any of the hardware devices attached to the system. To accommodate that possibility, Windows 95 sets up an I/O permission bitmap for each VM. The bitmap is an array of flags, one flag for each of the 386's I/O ports, that specifies whether the application can access the I/O port directly. If no access is granted—the normal case—the 386 signals a general protection fault whenever the application refers directly to the I/O port. For an MS-DOS application, a direct access attempt is not necessarily a program error, as it is for a Windows application. For example, a communications application will access the I/O ports for the COM device directly. For the application to run correctly, Windows 95 must allow this I/O port access to happen—assuming that some other program is not already in control of the same COM port. This whole treatment of virtual machine management and direct device control—referred to as *device virtualization*—is a key element of Windows 95. The most important aspect of device virtualization to note here is that the 386 provides the hardware facility for selectively protecting the I/O ports on an individual, program-by-program basis and informing the operating system each time a direct access occurs.

Virtual 8086 Mode

Without the virtual 8086 feature (most often called simply *virtual mode*), running MS-DOS applications under Windows 95 would be as difficult and error-prone as running them under OS/2 or Windows on the 286 processor. If you used earlier versions of either OS/2 or Windows on 286 systems, you'll remember both the errors and the major limitation: only one MS-DOS program could run at any one time. Clearly, I/O permission handling is a key requirement of the 386's virtual 8086 mode. A few other issues are important in Windows 95 running in virtual mode.

Virtual 8086 mode is an inherent part of the protected mode architecture of the 386. Programs running in virtual 8086 mode are running in protected mode. On the 286, MS-DOS programs didn't have a virtual mode (protected mode) to run under. To run an MS-DOS program on the 286, there was no choice but to run the processor in real mode. Real mode provided absolutely no memory and device protection, and what's more, the MS-DOS program had to occupy the first megabyte of the system's address space. The 386 solved all of these problems:

- Virtual 8086 mode execution remains subject to all the 386 memory and device protection rules. The operating system has control over the resources it allocates to the virtual mode program. The 386 reports to the operating system any attempted access to resources outside the allocated set.
- The operating system can load virtual mode programs anywhere in memory. The 386 translates virtual mode addresses using the 386 protected mode rules. All of the 386's paging capabilities are in play in virtual mode, so virtual mode programs running on the 386 can be swapped just as other protected mode programs can be.
- Unlike running an MS-DOS program on the 286 by means of a switch to real mode, running a virtual mode program on the 386 doesn't require a lengthy mode switch operation. Task switching between a Windows application and an MS-DOS application on the 386 is much faster than it was on the 286.

Setting up a virtual mode program on the 386 is straightforward. Once the program is loaded, the operating system simply identifies it as a virtual mode program by setting a single flag in one of the 386's control registers. The 386 then imposes the rules of 8086 program execution on the virtual mode program. Specifically, registers are 16 bits only (not 32 bits) and addresses are 20-bit values generated exactly as they would be on an 8086. Of course, this is only half the story. Emulating an 8086 processor is one thing. Emulating an entire PC, including MS-DOS, is entirely another. That problem has been passed along to Windows 95 to solve.

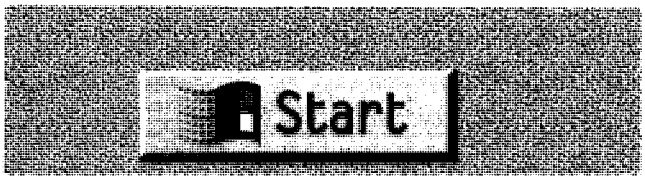
Conclusion

The Intel microprocessor has accumulated enormous capability since its simple beginnings with the introduction of the 8080 in 1974. In a scant twenty years, the microprocessor has matched or surpassed the capabilities of any mainframe processor costing thousands of times more. Along the way, the designers at Intel have had the good fortune to be able to learn from one failed experiment in protected mode—the 80286—and get it right the next time. The 80386 architecture, particularly its support of virtual 8086 mode within a paged virtual memory

scheme, has proved to be the right platform for building today's advanced 32-bit operating systems. The successor processors, the 80486 and the Pentium, have adopted the same basic architecture without change, and it's a sure bet that successors to the Pentium will do the same.

Windows 95 takes full advantage of all of the 386's capabilities. There's a lot going on under the hood when you run applications on Windows 95. Fortunately, neither the user nor the application programmer has to pay much attention to Windows 95's system and program management activities. This is as it should be.

That was the basics of how the hardware works. Now for the software. It's time to look at Windows itself.



CHAPTER THREE

A TOUR OF CHICAGO

In this chapter, we're going to take a tour through Windows 95—looking briefly at the structure of the system and the associated terminology. You may know Windows intimately already, in which case there'll be sections of this chapter that you'll skip through quickly. Chapter Four is where the detailed examination of Windows 95 begins. The goal for this chapter is to give you a sufficient grounding in the Windows system so that you can approach the new material in Chapter Four with ease. Although a lot of the information in this chapter is common to both Windows 3.1 and Windows 95, it will be Windows 95 that we dissect. Even if you've spent the last few years disassembling the several versions of Windows, you may want to flip through this chapter to make sure that my terminology matches yours and to get a quick overview of the structure of Windows 95.

Here's what we're going to look at in this chapter:

- The structure of the Windows system, including the graphical components of Windows and the system's support for Windows applications and MS-DOS virtual machines
- The Windows multitasking model
- The elements of the Windows user interface
- Some aspects of Windows application programs

System Overview

Over the course of successive version releases, Windows has grown from its original role as a graphical extension to MS-DOS to encompass many of the functions of a full operating system. From its very first release, Windows handled program loading functions. With Windows 95, the transformation is complete. Windows is now a complete operating system

with MS-DOS compatibility built in. The Windows 95 "single application mode" allows you to run MS-DOS as a fallback operating system if you want to run an application that can't function under Windows.

Figure 3-1 shows a block diagram view of the major components of Windows 95. Let's look at these components in a little more detail.

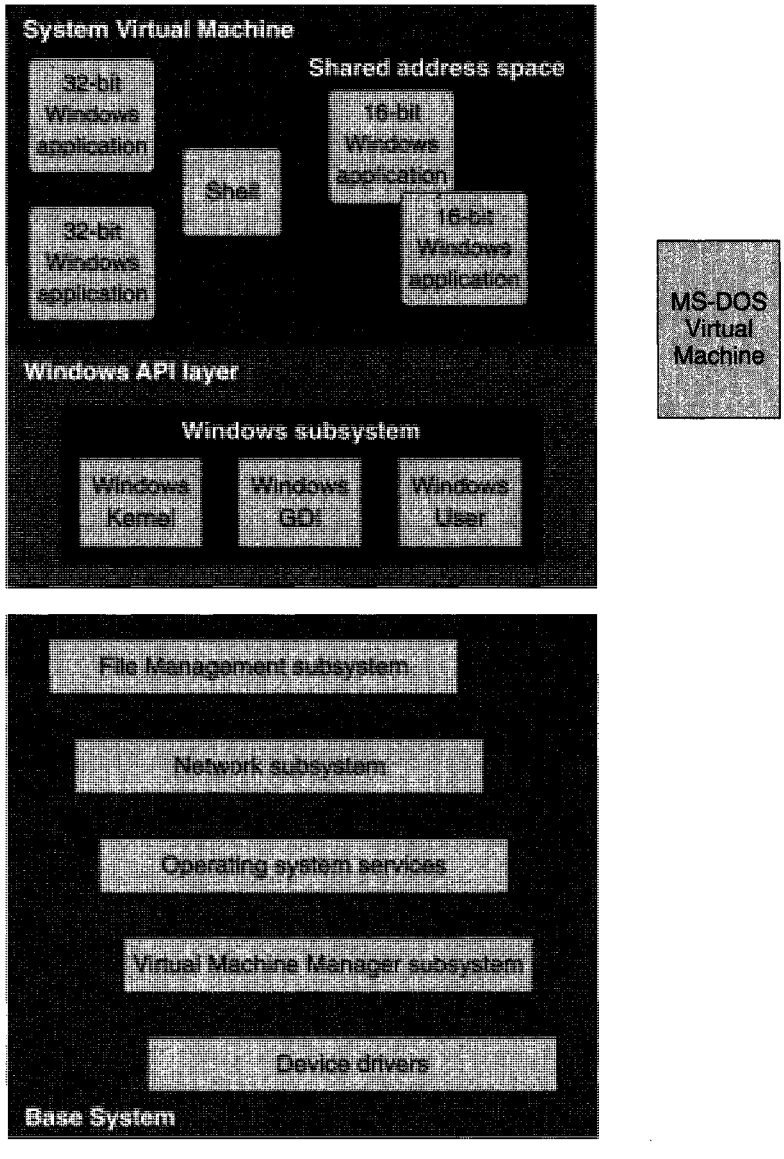


Figure 3-1.
Windows 95 system architecture.

The **System Virtual Machine** (or simply *System VM*) is the name given to the environment in Windows 95 that supports all the Windows applications and the Windows subsystem components such as the Graphics Device Interface (GDI).

32-bit Windows applications are the new Windows applications that use the 32-bit memory model of the 80386 processor and a subset of Microsoft's Win32 application programming interface (API). In Windows 95, each of these so called *Win32 applications* has a private address space that's inaccessible to other applications. 32-bit applications can be preemptively scheduled by Windows 95.

The **Shell** is a 32-bit Windows application that provides the essential user interface to the system. The Shell in Windows 95 consolidates the functions of the Windows 3.1 Program Manager, File Manager, and Task Manager utilities into a single application.

16-bit Windows applications are the "older" Windows applications, the ones you use on Windows 3.1 today. These applications use the segmented memory model of the Intel processor family—really an 80286 memory model. As in Windows 3.1, the 16-bit applications running under Windows 95 share a single address space and can't be scheduled preemptively. You'll hear Microsoft refer to these applications as *Win16 applications.*

The **application programming interface layer** in Windows 95 provides full compatibility with the existing Windows 3.1 API as well as support for the new 32-bit API accessible only to 32-bit Windows applications. The 32-bit API is a subset of Microsoft's full Win32 API first seen in Windows NT and in the Win32s add-on for Windows 3.1.

The **Windows Kernel** supports the lower-level services required by Windows applications, such as dynamic memory allocation. For Windows 95, the Kernel provides these services to both 16-bit and 32-bit applications.

GDI is the core of Windows' graphical capabilities, supporting the fonts, drawing primitives, and color management for both display and printer devices. Although GDI in Windows 95 continues to support existing 16-bit applications, it includes significant new features available only to 32-bit programs.

User is the window manager—the Windows 95 component that manages the creation and manipulation of on-screen windows, dialogs, buttons, and other elements of the Windows interface.

MS-DOS Virtual Machines support the execution of MS-DOS applications under Windows. As in Windows 3.1, the user can run multiple MS-DOS VMs concurrently. Windows 95 includes several new features designed to improve the user's management of these VMs, but the basic design for MS-DOS VM support hasn't changed a great deal.

The Base System

The remaining modules implement various aspects of the underlying operating system in Windows 95. The collection of these components is usually referred to as the *base system.*

File management has changed dramatically in Windows 95. In Windows 3.1, it's MS-DOS that controls the local hard disk filesystem. This MS-DOS control impaired the performance of Windows, and the opportunity to improve filesystem support didn't really exist while MS-DOS remained in control. Under Windows 95, the situation is entirely different. Notably, MS-DOS is no longer used for the management of files on local disks.[1] The new file management subsystem provides a series of interfaces that allows all local disk filesystems (including the CD ROM filesystem) and multiple network filesystems to coexist.

The **network subsystem** is the latest incarnation of Microsoft's peer-to-peer network first seen in the Windows for Workgroups product in 1992 and later seen in Windows NT.[2] The network subsystem uses the new file management subsystem to coordinate its access to remote files. Other network suppliers can also plug their products into the new file management services, allowing a user to simultaneously access more than one type of host network. Windows provides built-in support for SMB, Novell, and TCP/IP protocols.

1. As we noted in Chapter 1, there may yet be a version of MS-DOS that also includes the new filesystem capabilities. But it won't be the MS-DOS we're familiar with.

2. As of July 1994, it isn't clear how Microsoft will package the Windows 95 networking features. They might all be in the same box as Windows 95, or they might not.

Operating system services in Windows 95 include major components such as the Plug and Play hardware configuration subsystem as well as a miscellaneous collection of functions such as those that fulfill date and time of day requests.

The **Virtual Machine Manager** is the heart of the Windows 95 operating system. It includes software to implement all the basic system primitives for task scheduling, virtual memory operations, program loading and termination, and intertask communication.

Device drivers in Windows 95 can come in a number of different forms—real mode drivers and so called virtual drivers, or VxDs, among others. Some systems may still require the use of older real mode MS-DOS device drivers to support particular hardware devices, but one of the development goals for Windows 95 has been to develop protected mode drivers for as many popular devices as possible, including new protected mode drivers for the mouse, CD ROM devices, and many hard disk devices.

Virtual device drivers, or **VxDs,** take on the role of sharing a single hardware device among several applications. For example, running two MS-DOS applications in separate screen windows requires the system to create two MS-DOS VMs each of which wants access to the single physical screen. The screen driver VxD has to support this sharing requirement. "VxD" is also used as a general descriptor for other 32-bit operating system modules.[3]

Windows and Modes

You may never have run Windows on anything other than a 386-based system with a decent amount of memory—in which case, you've probably only ever used Windows in its *enhanced mode.* Operationally speaking, this meant that Windows used all the capabilities of your 386 processor, including demand paging and virtual 8086 mode. If your history with Windows goes back further, to 286- and even 8088-based systems, you will have heard the terms *real mode* and *standard mode* applied to Windows. If you knew those terms then, forget them now. Windows 95 operates only in enhanced mode. In fact, there is no longer a term "mode" for Windows.[4]

3. "VxD" actually stands for "Virtual anything Driver."

4. With Windows 95, support for the EGA as a display adapter also disappears. A Windows capable machine now requires at least a 386SX processor, 4 MB of memory, and a VGA.

Virtual Machines

The word "virtual" appears everywhere as a qualifier for terms in Windows 95.[5] Indeed, the provision of a virtualized environment for the execution of application programs is a key to many of the capabilities of Windows 95. The most important of the "virtual" features is undoubtedly the support for the *virtual machines* that host the running programs, so it's important to understand both the associated terminology and the technical basis for Windows virtual machines.

It's easy to get confused about virtual machines. Intel uses the term *virtual 8086 machine* to describe the use of the virtual 8086 processor mode to emulate an Intel 8086 processor on the 80386. This virtual 8086 machine includes the 1-megabyte address space, the CPU registers, and the I/O ports. A *Windows virtual machine* (usually called simply a *Windows VM*) refers to a *context* for the execution of an application program. A VM context includes the application's map of addressable memory and the contents of the hardware registers as well as the Windows resources allocated to the application. Because under Windows 3.1 every Windows VM runs at least part of the time in the hardware virtual 8086 mode (which is still a protected mode), there are abundant possibilities for misunderstanding. Many books and articles about Windows fail to distinguish among the many possibilities when they use the term "virtual." A Windows VM is not the same as an Intel virtual 8086 machine. Here's what's important about Windows VMs:

- Windows VMs are either *MS-DOS VMs,* each of which runs a single MS-DOS session, or a *System VM* that provides the execution context for all Windows applications.
- The System VM runs in protected mode all the time. Under Windows 3.1, there comes a point at which the System VM switches from protected mode to virtual 8086 mode so that MS-DOS code can run. This very rarely happens in Windows 95.
- Windows uses virtual 8086 mode to run MS-DOS applications. The system uses the processor's virtual 8086 mode to erect a controllable shield around code that would otherwise need to execute in real mode.

5. The marketing slogan chosen for the original introduction of Windows/386 was "Virtually Everything." It's a tagline that still seems to be appropriate.

- Windows applications on Windows 95 never use virtual 8086 mode. They execute in protected mode all the way down to the bare hardware.[6]
- An MS-DOS VM is a Windows VM running an MS-DOS application in virtual 8086 mode.
- Notwithstanding their association with virtual 8086 mode, MS-DOS VMs can run in 32-bit protected mode under Windows with the mediation of a DOS extender that conforms to the DPMI interface. When an MS-DOS VM switches to protected mode, it's no longer running in the processor's virtual 8086 mode, but Windows still considers it to be an MS-DOS VM. (This is a subtlety that's rarely recognized.)

To make things potentially more confusing, the word "virtual" is also used in talk about memory addresses. In Chapter Two, we looked at the details of how the 386 translates virtual addresses, generated by an individual program, to physical addresses that reference actual memory locations. Software running in any Windows VM always generates virtual addresses. The system itself uses virtual addresses. The only time that physical addresses come into play is when the memory management subsystem sets up the processor's page tables to provide the mapping between virtual and physical addresses.

- At least in this book, "address" and "virtual address" are synonymous. The term "physical address" will mean exactly that.
- An MS-DOS VM usually has an address space covering addresses from 0 to 1 megabyte. This is a virtual address space. The system maps this virtual address to its chosen set of physical addresses using the 386's virtual memory capabilities. The pages of the virtual address space could be widely scattered in physical memory.
- The System VM can have a much larger virtual address space than an MS-DOS VM running in virtual 8086 mode. Applications running in the System VM run in protected mode and can make use of this large virtual address space.

6. This isn't strictly true since Windows 95 still runs MS-DOS device drivers in virtual 8086 mode if there's no protected mode driver available. But real mode drivers are an endangered species.

Windows Virtual Machines

Regardless of whether it's an MS-DOS VM or the System VM that contains all the Windows applications, you define the capabilities and current context of a virtual machine by looking at the resources allocated to it. Each VM has to include the following:

- A memory map that defines the virtual memory accessible to the currently executing code within the virtual machine.
- An execution context, defined by the state of the VM's registers (the directly accessible CPU registers as well as other controlling factors such as the CPU privilege level).
- A set of resources accessible to the application running within the VM. Within the System VM, every Windows application accesses resources using the Windows API. In an MS-DOS VM, an application uses the MS-DOS software interrupt (INT) interface and may also try to access the hardware directly.

The virtual machine environment of Windows 95 remains heavily reliant on the underlying capabilities of the 386. The 386 dependence offers advantages:

- The virtual memory allocated to each VM is separated from the virtual memory allocated to other VMs. Each MS-DOS VM runs in a private address space, unable to interfere with applications running in other MS-DOS VMs or in the System VM.
- The memory and I/O port protection capabilities of the 386 allow every device on the system to be completely protected. Any MS-DOS application can run, convinced that it has the whole machine to itself and ignorant of the fact that it might actually be sharing the host system with other MS-DOS VMs or Windows applications.

Initialization

During initialization, the operating system sets up the System VM and prepares the *global context* for all MS-DOS VMs. Under Windows 3.1, this is essentially a snapshot of MS-DOS just at the point at which the user types the *win* command. Subsequently, whenever the system creates a new MS-DOS VM, this global context is used as the basis for the new

VM's context. The snapshot includes all TSRs, environment variables, and so on. Windows 95 is subtly different from Windows 3.1 during this initialization phase. With Windows 3.1, it's up to the user to enter the *win* command and start the initialization of the Windows system. Windows 95 immediately gains control and switches to protected mode to complete the initialization process after loading—no *win* command is needed. In either case, when Windows switches to protected mode, it pushes the real mode code aside and takes control of the machine. Windows 95 still processes the CONFIG.SYS and AUTOEXEC.BAT files if they exist, so the user can still customize the global MS-DOS context by including commands in these two files.

The System Virtual Machine

The context for the System VM is a protected mode environment in which all the Windows applications run, together with the major components of the Windows graphical subsystem. The interface between any application and Windows is by means of one of hundreds of *application programming interface* (*API*) functions.[7] This type of interface allows applications to request system services using named function calls rather than the numbered software interrupt scheme used in MS-DOS applications. The linkage between a Windows application and the functions in the Windows subsystem is made at program load time by means of a technique called *dynamic linking*.

Windows 95 introduces support for a new class of applications: the 32-bit applications that use the Windows 95 subset of Microsoft's Win32 API. These 32-bit applications run within the System VM context, but each has a private protected address space that prevents other applications from accessing its private memory.

Windows 3.1 relies upon cooperative multitasking as the basis for its task scheduling. Under Windows 95, cooperative multitasking is still the basis of task scheduling for the older 16-bit applications. However, the system schedules Win32 applications using a preemptive scheduling algorithm. For the user of a system that runs Win32 applications only, the preemptive scheduling means faster and smoother response when several applications run concurrently.

A Windows program relies on the system to deliver a stream of *messages* to it to inform it of new events—mouse clicks in one of the

7. As of early 1994, one rough count had the number of Windows 95 APIs, messages, and macros totaling well over 2000.

application's windows, new programs starting up, and so forth. Under Windows 3.1, the system uses a single queue to hold all the messages that originate within the system. As a result, it's possible for one errant application to choke the flow of messages to all the applications. Windows 95 provides for the system to put messages destined for Win32 applications into private message queues, reducing the possibility of the system's grinding to a halt when one application fails to service the message queue.

Windows 3.1 relies upon MS-DOS for filesystem access. Although this is about the only significant reliance on MS-DOS within Windows 3.1, it is a weak point of the system. This remaining dependence on MS-DOS for filing support creates a whole catalog of problems that the Windows designers have grappled with over the course of several releases. They finally fix the problems in Windows 95 by replacing the MS-DOS filesystem services with a new protected mode subsystem.

All MS-DOS filesystem services are accessed by means of the INT 21H software interrupt. Within the System VM itself, the execution of the INT 21H instruction causes a general protection fault that the operating system catches and handles. Windows 3.1 deals with this fault by arranging for the System VM to switch temporarily to virtual 8086 mode so that the MS-DOS INT 21H code can execute correctly. Once the file operation is completed, the System VM returns to protected mode and the Windows application code continues to execute.

Windows 95 catches the same fault and simply hands it to the protected mode filesystem manager for processing. No switch from protected mode to virtual 8086 mode occurs, and providing there is a protected mode device driver in use for the target device, the System VM context remains a protected mode context throughout the entire operation.

MS-DOS Virtual Machines

An MS-DOS VM is a faithful replication of a PC running MS-DOS. As far as the application is concerned, the VM has a megabyte of memory with a memory map corresponding to the hardware memory map. For example, the directly addressable video display memory is at memory address B8000H. The context for the MS-DOS VM is usually, though not always, a virtual 8086 mode environment with a copy of MS-DOS mapped into the virtual address space of the VM.

Applications in an MS-DOS VM will use the software interrupt services of MS-DOS (predominantly the INT 21H services) to make system

requests. Under Windows 95, these requests ultimately pass to the protected mode code that implements the system services. In the case of filesystem requests, the INT 21H call will be passed to the new filesystem manager to be handled together with other concurrent requests from applications running in the System VM.

MS-DOS VMs are set up using a VM that you never see—unless you start poking around with a debugger—and it's a VM that never contains an application that actually runs. This is the VM that is set up with the initial state of the MS-DOS environment once system booting and the processing of CONFIG.SYS and AUTOEXEC.BAT are complete. Within this hidden VM is everything that is global to the MS-DOS environment. For example, if your AUTOEXEC.BAT runs a TSR program before it starts Windows, that TSR program will be loaded and will become part of the global MS-DOS environment. Even under Windows 95, where there's less reliance on MS-DOS, you can still use CONFIG.SYS to load device drivers and AUTOEXEC.BAT to load TSRs as parts of the global MS-DOS environment.

Once this global initialization is complete, Windows needs somewhere to save a snapshot of the MS-DOS environment. It sets up the hidden VM context to be used as the initial state of every MS-DOS VM that's subsequently started. The saved hidden VM itself never runs. Later on, when you start an MS-DOS application from within Windows, the system creates a new MS-DOS VM—meaning that it allocates some memory and the appropriate control blocks within the system—and then copies into the new VM the entire global environment from the hidden VM. This copying means that the initial state of the new MS-DOS VM is exactly the state you'd achieve if you had just turned the machine on and run through the startup procedure again. This copying from the hidden VM also explains why changes that you make in one MS-DOS VM don't affect any of the others—either those already running or new VMs that you run later. To verify this inviolability of the MS-DOS VMS, simply run a few MS-DOS VMs and change the command prompt in each—local changes won't affect the saved global VM context that governs the initial states of all the VMs.

Protected Mode MS-DOS Applications

One complexity that the Windows designers have had to deal with is the fact that MS-DOS applications are not simply real mode applications anymore—they can also run in protected mode. You can trace

this wrinkle back to a few years ago when the hunt for more than 640K of memory began in earnest. Expanded memory, extended memory, high memory, and the products that exploited them—such as Quarterdeck's QEMM—became popular resources. For a while, the whole situation was a mess, with various designs jockeying for position as the standard.

One group of vendors sought order by agreeing to the VCPI (Virtual Control Programming Interface) specification. VCPI was pretty good except that it didn't fully support Windows. So after a brief face-off with Microsoft, vendors came up with the DPMI (DOS Protected Mode Interface) specification. Programs that conform to the DPMI specification can run under MS-DOS and Windows and can exploit protected mode on both 286 and 386 systems.

DPMI

The DPMI specification lays out the definition of an MS-DOS software interface that ultimately allows MS-DOS applications to exploit the 32-bit protected mode while running under Windows. DPMI actually allows low-level software components called *DOS extenders* to coexist with Windows. A DOS extender supports the execution of protected mode programs that want to call on MS-DOS for file I/O and other services. The need for the DPMI specification became apparent during the development of Windows 3.0, when Microsoft and other companies embarked on parallel efforts to provide support for 32-bit protected mode program execution. Microsoft's interest was in Windows, since Windows is itself a DOS extender. It was clear that there would be a number of DOS extenders on the market, so vendors developed DPMI as a way of allowing them to coexist. Today you can find DOS extenders in use in several kinds of popular applications that need more than 640K of MS-DOS memory: compilers, database programs, and others. The interfaces to the various DOS extenders are not standardized—the DPMI interface that allows the DOS extenders to coexist with Windows is.

The DPMI–DOS extender exploitation of protected mode is essentially the best way to allow an MS-DOS program to get at more memory and to use 32-bit addressing (as opposed to struggling on with segmented addressing). Windows 3.1 implements DPMI and DOS extender functionality within a single module, so as far as a Windows programmer is concerned, the DPMI and extender services are indivisible. This architecture does allow a user to start MS-DOS VMs that run

applications that make use of alternative DOS extenders rather than Windows itself as a DOS extender. In that scenario, Windows provides only the DPMI services.

The DPMI specification defines two software components needed to provide a full implementation. The *DPMI Host,* or *DPMI Server,* is the lowest-level software component responsible for administering the DPMI services. All the DPMI functions are available by means of a call to INT 31H with a function number that identifies the particular DPMI service that's required. These services really are very low level—the allocation of descriptors within the LDT or GDT and the reading and writing of MS-DOS interrupt vectors, for example.

The *DPMI Client* is any program requesting DPMI services, usually the DOS extender. Although it's possible, the DPMI interface is not intended for direct use by application programs. It's up to the client to check for the presence of a DPMI server before any attempt to call the server is made. Most DOS extenders define a private API that allows a modified MS-DOS application to call the extender for protected mode services and to provide MS-DOS services to the application while it executes in protected mode.

Multitasking and Scheduling

One of the more complex Windows activities is its allocation of the processor to multiple programs. For a program to do anything, it has to execute instructions. Since Windows allows you to run several programs at once, there has to be a way of sharing the processor among these programs. Enter multitasking—and with it a great deal of terminology and debate.

Since so much terminology is associated with the subject of multitasking, we'll need to define a few terms in this chapter. Some of the terms are frequently used in both a generic context and a very particularized context. The word *task,* as we'll see, is a classic example. Windows is, generically speaking, a multitasking system, and a Windows 3.1 task is a very precise concept, represented by specific data structures and operational rules.

In the next chapter, we'll look at the details of the Windows 95 multitasking model. In this section, we'll give the subject a general review with a Windows bias.

Multitasking Models

The generic term *multitasking* refers simply to an operating system's ability to share the CPU among several programs. Most operating system designers refer to a program in its running state as a *task,* so you can think of a task as a program loaded into memory and actually doing something. The Windows NT and UNIX worlds both use the term *process* to mean the same thing. Windows 3.1 says *task* and, occasionally, *process.* And lo and behold, the word *process* is the term in favor for Windows 95. The term *task* has been officially removed from the Windows language. The term *process* is therefore what we'll use. Really, you can think of *task* and *process* as synonyms.[8]

As soon as you run Windows 3.1, you're multitasking since you're running the Program Manager and a number of other tasks that are actually part of the system itself rather than programs with visible windows on your screen. Windows 95 is no different in this respect. A few years ago, when observers first began to discuss multitasking operating systems for PCs, you often heard comments to the effect of "I don't need multitasking. I do only one thing at a time anyway." Unfortunately, people rarely understood that a multitasking system could offer features such as background print spooling and network connectivity even if the user only ran Lotus 1-2-3 all day. Nowadays good multitasking is considered to be essential to providing an effective environment for the PC user. Even if you only run Lotus 1-2-3/W all day long, Windows multitasking enables you to manage your network connection, the Print Manager, and your communications session at the same time.

The operating system component that manages the multitasking in both Windows 3.1 and Windows 95 is the *scheduler.* The scheduler deals principally with *time* and *events.* A Windows 95 process gets a *time slice* that determines how long it can use the CPU. At the end of the process's time slice, the scheduler decides whether to let a different process use the CPU.[9] Events influence the scheduler's decisions. To the scheduler, a mouse click is an event that may mean handing the CPU to the process that owns the window in which the mouse click occurred.

8. At this point you probably think this discussion is becoming very arcane. Unfortunately, *process* has a precise meaning in Windows and the lack of rigor with respect to such a term in most Windows documentation can generate considerable misunderstanding.

9. Unlike Windows NT, Windows 95 doesn't (and won't) support multiprocessor systems, in which the scheduler has more than one processor to allocate to processes.

Or the scheduler may consider the simultaneous completion of a network data transfer to be an event worthy of more attention than the mouse click. In that case, the process managing the network would get the CPU, and the other process would have to wait.

You'll hear Windows 3.1 described as a *cooperative* multitasking system and Windows NT described as a *preemptive* multitasking system. Cooperation and preemption are process scheduling techniques, and Windows 95 uses both of them, so we have to understand them. Preemptive scheduling puts the operating system in complete control over which process runs next and for how long. At any time, the scheduler can take the CPU away from the current process and hand it to another one. Typically, such a preemptive act will occur in direct response to an event that demands swift attention. The scheduler associates a *priority* with each running process. If an event occurs that is of interest to a high-priority process, the scheduler will preempt the current process and run the high-priority process. The scheduler gets control of the system either when a process surrenders the CPU (it reaches a point at which it's waiting for the user, for example) or when there's a clock interrupt. Most systems will program the clock to tick between 20 and 50 times a second, and the final tick is when the scheduler gains control and can preempt a running process.

Process priorities are recalculated frequently. For example, if the system has to choose between just two processes—one with a low priority and one with a higher priority—the low-priority process will never be able to run if the scheduler doesn't dynamically adjust the priorities. The duration of the time slice plays into the calculation of priorities as well. It makes no sense to continually give the CPU to a process and then preempt the process after it has executed only a few instructions. All that will ever get run is operating system code, not your spreadsheet or compiler.

Cooperative multitasking relies upon application programmers to help keep the system running smoothly. In the cooperative technique, the scheduler can switch processes only when the currently running process surrenders the CPU. If the current process decides to recalculate π to 5000 decimal places, there's nothing the scheduler can do about it. Good programming practice for cooperative multitasking systems dictates that applications should regularly hand the CPU back to the operating system—a technique called *yielding*. An application's yielding allows the scheduler to run a higher-priority process if one is ready. In Windows 3.1, cooperative multitasking is why no amount of

mouse clicking will help you when the current application has the hourglass cursor up on screen. The system duly registers all the mouse click events and adds them to the application's message queue, but until the current process surrenders the CPU, the scheduler can't switch away from it and allow another process to handle the new events.

Windows 3.1 is as insistent as it can be about getting applications to yield control of the processor. Essentially, every time an application calls the system, asking to deal with the next event, the system suspends the process and allows the scheduler to reevaluate process priorities. The lack of preemption doesn't make this way of handling the cooperative multitasking problem foolproof, however.

The absence of preemption in Windows 3.1 does make a number of design decisions easier for both operating system developers and application programmers. Neither has to worry about the operating system code's being reentrant, for instance. The system design doesn't have to account for the possibility of process preemption while system code is executing. Suppose, for example, that you run two Windows applications, both of which occasionally use a COM port to dial out and retrieve data from an information service. If one application could be preempted in favor of the other partway through the opening of the COM port, the OS would have to protect itself from the possibility that the second application would also start an open request. With no preemption, the OS doesn't have to worry: the first open request will always run to completion before the other application can run.

Ultimately, though, the lack of preemptive scheduling leads to problems. High-priority events can't be handled rapidly because an application won't relinquish the processor in time, for example; or an application that crashes will lock up the whole system because the operating system will be unable to deliver messages to other applications. MS-DOS itself has to have a nonpreemptive scheduling environment. MS-DOS knew nothing of multiple processes when it was designed, and despite the herculean efforts of many software developers to build multitasking systems on top of MS-DOS, there have always been shortcomings in the resultant products. Windows has been no exception to this nonpreemptive rule. Preempting MS-DOS at the wrong time can lead to disaster, so over the years the Windows designers have had to put up with building most of an operating system on top of a very unsuitable foundation. Windows 95 changes that.

Critical Sections

You'll hear programmers use the term *critical section* when they talk about developing software for any preemptive multitasking system. A critical section is a sequence of instructions executed by more than one process that for one reason or another must not be preempted before it completes execution. An obvious example of a critical section occurs during memory allocation.

Windows, along with most other operating systems, uses derivatives of thirty-year-old algorithms for keeping track of blocks of available memory. (It's not that the algorithms are outdated. It's just that they're as good as they ever need to be.) One particular algorithm in question maintains available memory blocks as a linked list, with a descriptor for each block that identifies its size and location. When Windows tries to satisfy an application's request for memory, it has to unlink the block from the list of available blocks.

At some point during the unlinking procedure, the list data structure is in a mess, with invalid pointers or erroneous flag bits set. If the system were to reschedule right at that point, a different process might initiate a new memory allocation request. Since the first process would not yet be complete, the new process would eventually stumble while trying to manipulate the invalid list data structure and probably crash the whole system. To guard against such a situation, the code manipulating the list maintains a critical section between the entry and exit points of the sensitive instruction sequence. Once the process enters the critical section, the system guarantees that the process will exit the critical section before any other process can enter it. This isn't to say that the system necessarily ignores other processes while a critical section is executing. For example, ignoring hardware interrupts during the execution of a lengthy critical section would be indicative of bad system design. Critical section management does guarantee, though, that once a process has entered a critical section, the system will suspend any other process trying to enter the same section.

The technique of allowing only one process at a time to execute a critical section is sometimes referred to as *mutual exclusion*, and the undesirable situation in which several processes fight to get at a protected resource such as memory by entering the critical section is called *contention*. The Windows Virtual Machine Manager has long supported critical section management for device drivers. Preemptive scheduling

means that Windows 95 has to support similar critical section management functions at the API level. The newly improved nature of multitasking and preemption in Windows 95 means that you'll hear more frequently about objects called *mutexes,* or *semaphores,* that are used to control process entry and exit of critical sections.

Processes in Windows

So, amidst a collection of virtual machines and in a system that supports cooperative multitasking, what exactly is a process in Windows 95? It is one of two objects:

- Windows considers each MS-DOS VM to be a single process. Regardless of what's going on inside that VM, to Windows it is only one process.
- Each executing Windows application is also a process. Remember that every Windows application runs within the System VM, so this view of the System VM as containing multiple processes points up another difference between the System VM and an MS-DOS VM.

Under Windows 3.1, all of these processes are described within a system data structure called the *Task Database,* or *TDB* for short. Windows 3.1 actually identifies an MS-DOS VM process by marking the appropriate TDB entry as being the WinOldAp application.[10]

Under Windows 95, the tasking model is considerably more complex. The most important change from the application developer's point of view is the addition of *threads* to the system. Under Windows 95, threads rather than processes are the objects managed by the system scheduler. A thread defines an execution path within a process, and any process can create many threads, each of which shares the memory allocated to the original process. Multiple threads allow a single application to easily manage its own background activities and to offer a highly responsive interface to the user.

Modules

In Windows, the term *module* describes a related collection of code, data, and other resources (such as bitmaps) present in memory. Typically,

10. WinOldAp is the name given to the entity that controls a single MS-DOS VM. You'll see the name in various Windows status displays and documentation items.

such a collection will form either a single application program or a dynamic link library. Windows maintains a data structure, known as the *module database,* that identifies all the modules currently active in the system. The module database describes an essentially static collection of objects rather than the dynamic collection referenced by the task database.

Keeping a record of currently loaded modules is important because such a record is the basis for the resource sharing supported by Windows. The second time you run the WordPad (née Notepad) application, for example, Windows can see that the code segments and the bitmap that forms the icon are already in use. Rather than loading a second copy and consuming more memory, Windows simply creates additional references to the resources already in use.

During the life of the system, Windows maintains a usage count for each resource. As applications make use of a resource, the system increments the reference count. When the application terminates, the system reduces the reference count. A reference count of 0 is the indication that the resource is no longer in use and that the system can remove the resource and reclaim the memory it occupied.

API Support

The Windows 95 API coverage is, to say the least, extensive. The Windows 95 API includes a subset of Microsoft's Win32 API and provides compatibility by including support for 16-bit Windows applications and MS-DOS applications. Microsoft recommends that 16-bit Windows application development cease with the introduction of Windows 95 and, to encourage developers to make that choice, makes the new capabilities of the Windows 95 system accessible only to 32-bit applications. The mere opportunity to finally abandon the Intel architecture's segmented memory model is likely to be enough reason for most developers to switch. Add in the enhancements available to Win32 applications, and switching becomes a pretty attractive option.

Windows supports its APIs by means of three major components: Kernel, User, and GDI. Kernel incorporates the most operating-system-like functions—memory allocation, process management, and the like. The User module focuses on the window management issues that come up throughout Windows operation: window creation and movement, message handling, dialog box execution, and a myriad of related functions. GDI is the Windows graphics engine, supporting all the line drawing, font scaling, color management, and printing capabilities of the system.

Every Windows application shares the code in these three modules. In Windows 95, Kernel, User, and GDI have each a 16-bit and a 32-bit implementation resident in the system. And a lot of code is shared between, for example, the 16-bit and the 32-bit implementations of GDI. Applications don't have to take any special note of this dual existence, though. The system connects the application with the appropriately sized subsystem.

Each Windows API function is accessible by means of a name—in contrast to the MS-DOS API scheme of numbered interrupts. To get an application to call on one of the services in a Windows subsystem, the programmer simply uses the target function name in the application source code and compiles and links with the appropriate libraries, and the application is ready to run. This sounds normal so far, but if you examine the compiled program, you won't find any code that actually implements a Windows API function. If you're a C programmer, you'll have used the *printf()* function frequently. Poke through the compiled program, and sure enough, you'll find a stream of code and data that implements *printf()*, and the same is true for many other functions.

What you will find if you care to dissect a compiled Windows program is a collection of references to the Windows API functions—references that are necessary if Windows is to be able to load the application correctly. And think about that *printf()* example again—every program has its own copy of the code for *printf()* linked in, whereas the Windows program that calls *GetMessage()* calls the single copy of this function that resides in the User module. So does every other Windows program. In fact, the Kernel, User, and GDI modules are all examples of Windows *dynamic link libraries* (*DLLs* for short). Windows uses DLLs extensively, and the technique that allows an application to call a DLL is *dynamic linking.*

Dynamic Linking

Nowadays it's customary to rely upon the dynamic linking capabilities of the target operating system when preparing an application for execution. Windows and Windows NT have the capabilities, OS/2 has them, and so does UNIX. A compilation and link procedure used to involve the linker in scanning object code libraries and copying large amounts of code and data into the application's executable file. No more. In a dynamic linking environment, the traditional role of the linker is now split between the link step and the program loading step undertaken by the operating system.

The linker still scans a set of libraries. Some of the libraries include runtime support code that ends up in the executable file; others simply contain references to functions that won't be fully resolved until the operating system loads the program. In Windows, such libraries are called *import libraries,* and together they contain a defining reference for each and every Windows API function. The linker scans the import library and embeds in the executable file a target module name and a numeric entry point. If an application calls the Windows *MessageBox()* function, for example, the executable program file will include a reference to the User module entry point number 1. The application's calling the GDI *LineTo()* function will embed a reference to the GDI module entry point number 19. At program load time, it's the operating system's responsibility to replace these references with addresses that are valid for use in function calls. Any module that satisfies these references via dynamic linking is called a dynamic link library. Every DLL declares a set of entry points called *exports* that satisfies the external references.

Much of Windows itself is a collection of DLLs, and the system makes heavy use of the runtime name resolution capabilities to interconnect its various components. For example, printer device drivers support a standard set of entry points. When the GDI module calls a printer driver, it references a function that will be resolved via a runtime dynamic link. Regardless of what type of printer is involved, each printer driver supports the same set of entry points. Rather than relinking the operating system when you install a new printer, you simply replace the file containing the device driver code, and the new driver satisfies the same set of dynamic links. Figure 3-2 shows the first few entries for the dynamic links exported from the Windows 3.1 Hewlett-Packard PCL and PostScript printer drivers.

```
; "DDRV HP LaserJets and compatibles"

HPPCL . 1    DEVBITBLT
HPPCL . 2    COLORINFO
HPPCL . 3    CONTROL
HPPCL . 4    DISABLE
HPPCL . 5    ENABLE
    ...
```

Figure 3-2.
Dynamic link entry points in printer drivers.

(continued)

Figure 3-2. *continued*

```
; "DDRV PostScript Printer;100,200,300"
;
PSCRIPT . 1         BITBLT
PSCRIPT . 2         COLORINFO
PSCRIPT . 3         CONTROL
PSCRIPT . 4         DISABLE
PSCRIPT . 5         ENABLE
```

Notice that in each printer driver the names refer to functions within the driver. They could be any valid name. The external reference uses only the module name and the numeric identifier to resolve the dynamic link.

The Windows resource sharing technique also applies to DLLs. It has to—after all, DLLs are built for sharing. Loading unique copies not only is wasteful but also defeats the whole purpose of a DLL.

Support from the Base System

Ultimately, the Windows subsystem has to call on the services of the base system. This might be an explicit request—for example, to open a file. Or it might be an implicit one—for example, there's a page fault and the base system has to set about loading the missing pages from disk. In the case of an MS-DOS VM, the assistance of the base system is needed once the MS-DOS software interrupt executes.

A transition to the operating system code in the base system involves a transition between processor privilege levels. The Windows VMs usually run at ring three; the base system—the most privileged code in Windows—runs at ring zero. Chapter Four looks at the details of the transition to the base system code. The various ways in which it happens all amount to presenting the Virtual Machine Manager with an opportunity to gain control over the transition so that order can be maintained.

The base system code comprises a number of Windows *VxD*s. Although the name *VxD* and the term *virtual device driver* are used interchangeably, a VxD need have nothing to do with any hardware device. A VxD is simply a 32-bit protected mode module running at the processor's most privileged level of execution. Some VxDs do deal with hardware devices, and others supply operating system functionality that doesn't have anything directly to do with devices. The VxD architecture

was originally designed as a standardized format for 32-bit protected mode code modules. There is an API, internal to the base system, that VxDs can use.[11] Obviously, the scope of these functions is at a much lower level than the scope of the services called on directly by applications.

Memory Management

Memory management in Windows takes place at two different levels: a level seen by the application programmer and an entirely different view seen by the operating system. Over the course of different releases of Windows, the application programmer has seen little change in the available memory management APIs. Within the system, however, the memory management changes have been dramatic. Originally, Windows was severely constrained by real mode and 1 megabyte of memory. Then expanded memory provided a little breathing room, and currently the use of enhanced mode and extended memory relieves many of the original constraints. Windows 95 goes further yet and essentially removes all the remaining memory constraints.

Windows 95 continues to support all the API functions present in Windows 3.1, and you can still build and run applications that use the segmented addressing scheme of the 286 processor. However, if you look at the detailed documentation for the Windows 95 memory management API, you'll see that all of the API functions originally designed to allow careful management of a segmented address space are now marked "obsolete." The "obsolete" list includes, for example, all the functions related to selector management. The reason, of course, is the Windows 95 support for 32-bit linear memory and the planned obsolescence of the segmented memory functions—yet another unsubtle hint that the Win32 API is the API you should be using to write Windows applications.

Although use of the 32-bit flat memory model simplifies a lot of Windows programming issues, it would be misleading to say that Windows memory management has suddenly gotten easy.[12] Windows 95 actually has a number of new application-level memory management

11. The Windows Device Driver Kit is the best reference for detailed information on VxDs and the associated API functions.

12. The Windows 95 documentation lists 45 API functions under the heading "Memory Management." The "obsolete" list numbers 28 API functions.

capabilities. All of the functions relate to the management of memory within the application's *address space,* the private virtual memory allocated to the process. The systemwide management of memory is the responsibility of the base system, and the Windows API aims to hide many of the details of the system's lower-level functions.

Application Virtual Memory

Figure 3-3 illustrates the basic layout of a Win32 application's virtual memory. Every Win32 application has a similar memory map, and each such address space is unique. However, it is still not fully protected: the private memory allocated to one Win32 application can be addressed by another application. The Win32 application's private address space is also the region in which the system allocates memory to satisfy application requests at runtime.

The system address space is used to map the system DLLs into the application's address space. Calls to the system DLLs become calls into this region. Applications can also request the dynamic allocation of memory by means of virtual addresses mapped to the shared region. Having virtual addresses mapped to the shared address space caters to the need for controlled sharing of memory with other applications.

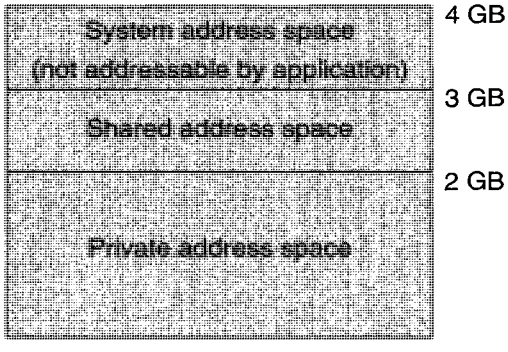


Figure 3-3.
Application virtual memory map.

Requests for memory at runtime fall into one of two categories: the application can make an explicit request for extra memory, or the system can respond to an implicit request for memory—that is, allocate memory to an application as a side effect of allocating some other resource. An implicit request occurs, for example, when an application

creates a new window on screen: the system must allocate memory for the data structures used to manage the window. Windows 95 claims memory for resource allocation from a large 32-bit linear region rather than from the restrictive 64K segment used in previous versions of Windows. An ongoing problem in versions through Windows 3.1, running out of memory during resource allocation, has been largely eradicated in Windows 95.

Heap Allocation

In Windows parlance, the term *heap* describes the region of memory used to satisfy application memory allocation requests. In Windows 3.1, the system maintains both a *local heap* and a *global heap*. The local heap is a memory region within the application's address space, and the global heap is a memory region belonging to the system. As an application makes requests for local memory, its address space is adjusted to encompass the newly allocated memory. The system resolves requests for global memory from the same system memory pool used for all applications. It's possible to run out of either or both resources, although the use of a 2-GB address space makes this highly unlikely. Exhaustion of the local heap affects only a single application. Exhaustion of the global heap has systemwide repercussions.

Windows 3.1 programmers have to consider a variety of factors as they decide how to satisfy an application's runtime memory requirements. Windows 3.1 also has a range of API functions for manipulating dynamically allocated segments, and the manipulation of these shifting regions is further complicated by the underlying segmented memory model. It isn't just a chunk of memory that must be allocated. The application also needs a selector so that it can address the memory correctly. Under Windows 95, the Win32 application model does away with all these considerations. Selectors are no longer required—it's simply a 32-bit address that identifies the new memory—and the local and global heaps are merged into a single heap. The API functions that deal with selectors and the manipulation of memory regions in a segmented model all become obsolete.

Windows 95 Application Memory Management

For a Windows programmer, the Win32 API greatly simplifies the most common dynamic memory allocation chores. Furthermore, the increased capability of the underlying 32-bit architecture allowed the Windows designers to add a number of new functions for application memory management.

- Windows 95 provides functions that support private heaps whereby an application can reserve a part of memory within its own address space. The application can create and use as many private heaps as it wishes and can direct the system to satisfy subsequent memory allocation calls from a specific private heap. An application might use the local heap functions to create several different memory pools that each contain data structures of the same type and size.
- Windows 95 provides functions that allow an application to reserve a specific region of its own virtual address space that once reserved won't be used to satisfy any other dynamic memory allocation requests. In a multithreaded application, the 32-bit pointer to this reserved region is a simple way to provide each thread with access to the same memory.
- Memory mapped files allow different applications to share data. An application can open a named file and map a region of the file into its virtual address space. The data in the file is then directly addressable by means of a single 32-bit memory address. Other applications can open the same file, map it into their private address spaces, and reference the same data by means of a single pointer.

System Memory Management

Regardless of changes in the details of application memory management, the Windows programming model has remained pretty consistent through the different product releases. Allocating blocks of memory at runtime, using a reference to a block to manipulate it, and ultimately returning the block to the system for re-use is the way in which Windows programmers have always dealt with dynamic memory requirements. Windows 95 is no different. What has changed, however, is the way in which the system realizes the application's requests for dynamic memory.

Starting with the Windows 3.0 enhanced mode and continuing with the Windows 95 Win32 application model, the Windows API manipulates only the application's virtual address space. This means that an application request for a block of memory will adjust the application's virtual address map but might do absolutely nothing to the system's physical memory. Remember that the 386 deals with physical

memory in pages each 4K in size. This page size is reflected in the virtual address space map of every Windows application. If an application requests 100K of memory, for example, its virtual address space will have 25 pages of memory added to it. The system will also adjust the data in its own control structures to reflect the application's new memory map.

However, at the time of allocation, Windows won't do anything to the physical memory in the system. It's only when the application starts to use the memory that the underlying system memory management kicks in and allocates physical memory pages to match the virtual memory references the application makes. If the application allocates but never references a region of its virtual memory space, the system might never allocate any physical memory to match the virtual memory. The ability of the 386 to allow physical memory pages to be used at different times within different virtual address spaces is the basis for the operating system's virtual memory capabilities.

Deep within the system are a range of memory management primitives available to device drivers and other system components that sometimes deal with virtual memory and sometimes force the system to commit actual physical memory pages. But these primitives are specific to the base operating system. Neither applications nor the Windows subsystem knows or cares about physical memory. Applications can force the system to allocate physical memory only by actually using the memory: namely, by reading from and writing to locations within a page. The separation of Windows memory management into the virtual and physical levels is a key aspect of the system. Applications and the Windows subsystems deal with defined APIs and virtual address spaces. The base system deals with physical memory as well as virtual address spaces.

Although physical memory is transparent to an application, its behavior can radically affect the performance of the system. For example, scanning through a two dimensional array of data row by row using C as the programming language will cause memory to be accessed from low to high virtual addresses because C stores two dimensional array data structures in *row major order.* As the memory sweep proceeds, the system will allocate physical memory pages to match the virtual memory accesses. Byte-at-a-time access will cause the system to allocate a new physical page every 4096 references. Other languages—FORTRAN, for example—store two dimensional arrays in *column major order.* Referencing the data row by row will generate memory references to widely scattered

memory locations, forcing a much higher frequency of physical page allocation and much-reduced application performance. So, although the programmer doesn't have to worry about matching virtual memory to physical memory, it is a good idea for the programmer to know something about how the underlying system primitives and hardware support the application.

Windows Device Support

The most important aspect of the Windows device driver architecture is its ability to *virtualize* devices. (Yes, it's that word again.) The greatest difference between the device drivers of Windows 95 and Windows 3.1 is the extensive use of protected mode drivers in Windows 95—in fact, it will be unusual if your system uses any real mode drivers at all after you install Windows 95. The use of protected mode for the drivers pays off in terms of both system performance and robustness. The manufacturers of disk devices can adopt a new driver architecture—borrowed from Windows NT—that almost guarantees the availability of a protected mode driver for every hard disk. In addition, new protected mode drivers for CD ROM devices, serial ports, and the mouse make the possibility of needing to support a device with a real mode driver quite remote.

Device Virtualization

The device virtualization capability allows Windows 95 to use the memory and I/O port protection capabilities of the 386 processor to share devices among the different virtual machines. Every MS-DOS VM believes it has full control over its host PC and is unaware of the fact that it might be sharing the screen with other MS-DOS VMs or with the Windows applications running in the System VM. For MS-DOS applications, the display drivers must reside in the lowest level of the operating system. Many MS-DOS applications, particularly those that use the display in a graphics mode or use serial ports, will address the hardware directly. Windows has to intercept all such direct access in order to bring order to a potentially chaotic situation. The MS-DOS application knows nothing of the need to cooperate with other applications and certainly doesn't depend on a system device driver to get the job done. With Windows applications, the system has a slightly easier task since device access is always

the result of a Windows API call. Thus, the operating system has control of the entire transaction, and the system components can collaborate as necessary.

You'll sometimes hear Windows device drivers referred to as *virtual device drivers* or even *VDDs*. But most of the time, a Windows device driver is classified as a VxD along with all the other VxDs that perform low-level system functions. Device drivers are written and built just as any other VxD is—usually in assembly language and always with the freedom to access any system data structure or memory location.

Minidrivers

The Windows device driver model has undergone some changes for Windows 95. The *minidriver* architecture first used for Windows 3.1 printer drivers and more recently for Windows NT disk drivers has found its way into the display and disk driver designs for Windows 95.[13] The principal idea of the minidriver design is to provide a single hardware-independent VxD that fulfills most of the necessary driver functions. This VxD interfaces closely with a minidriver whose role is to perform the hardware-dependent functions. Each minidriver consists of a set of the hardware-dependent functions called by the controlling VxD. Windows calls the central VxD, and when necessary, the VxD calls the minidriver.

This design offers a lot of advantages. The basic design tenet is that most drivers for a particular type of device contain roughly the same code. Re-implementing the same code for every slightly different type of device doesn't make a lot of sense—despite the fact that just about every operating system has done just that for years. Reducing the implementation task for a new device to simply developing a new minidriver helps everyone. The device manufacturer doesn't have to invest in writing code that already exists. The user can look forward to much higher quality drivers that are readily available when a new device first appears. Microsoft benefits since they can justify the investment of a lot more effort in the central screen VxD, for example, rather than have the dilution of the effort among drivers for dozens of slightly different VGA devices.

13. In Windows NT, disk drivers are actually called *port drivers*.

In the past, a counterargument always insisted that the minidriver model would degrade performance. This argument didn't work when it was applied to printers since the nature of the device makes it very slow in comparison to the processor anyway. Even the worst printer minidriver is probably fast enough to keep a printer fully occupied. Disk device minidrivers do require more attention to performance issues. However, a disk minidriver is a simple piece of code that shouldn't have a negative impact on performance if it's correctly written. Microsoft can provide lots of good examples to device manufacturers to make sure that disk minidrivers come out right. Screen devices are quite a different issue since performance under Windows is so critical. The importance of performance makes the adoption of a minidriver model for screen drivers an interesting design choice. Microsoft's confidence in its new display driver model comes from investing a lot of very talented effort in the central VxD.[14] Of course, it's still possible for a manufacturer to ignore the minidriver architecture and implement a device driver that bypasses the minidriver architecture. The manufacturer still has this option for supporting unusual devices or squeezing the last cycle of performance out of the device.

The Windows Interface

Let's review the major elements of the Windows user interface in preparation for an introduction in Chapter Five to the rather dramatic changes to be seen in Windows 95. If you're a Windows programmer, you're already intimately familiar with the user interface terms and the various user interface components. If you use Windows extensively, you've seen and used all of the major interface elements. However, while clicking your way quite happily through a complex dialog box, you may not have thought too hard about all the different elements that make up the dialog box.

What Is a Window?

Take a look at the Windows 3.1 screen shot in Figure 3-4. It's one of the more commonly used dialog boxes in Microsoft Word for Windows. You see it every time you print a document.

14. "World's fastest flat frame buffer device driver" is one claim. We'll see.

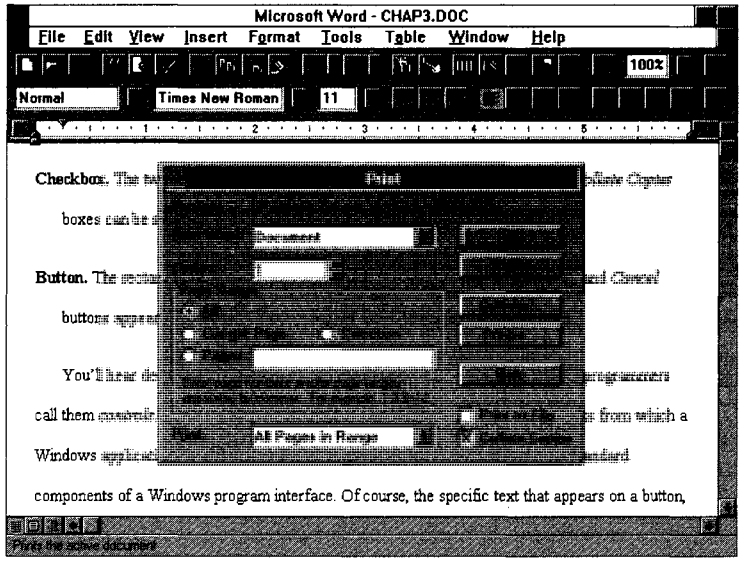


Figure 3-4.
Windows, windows, windows...

This dialog box actually contains several of the most common items used in dialog boxes—specifically:

Drop-down list box. The box to the right of *Print:*. Clicking on the arrow causes a list of items to appear from which the user can make a single choice.

Spin box. The box to the right of *Copies:*. Clicking the up and down arrows changes the numeric value in the box.

Radio buttons. The round buttons inside the *Range* box. The user can select just one of the *All*, *Current Page*, and *Pages* buttons. Clicking one of them causes the others to clear.

Checkbox. The two boxes at the bottom of the dialog box. The *Print to File* and *Collate Copies* boxes can be set on or off.

Button. The rectangular buttons at the right of the dialog box. The ubiquitous *OK* and *Cancel* buttons appear in almost every dialog box.

You'll hear designers refer to each of these interface items as *visual elements;* programmers call them *controls.* These and several other common elements are the building blocks from which a Windows application developer will assemble the various dialog boxes and other standard components of a Windows program interface. Of course, the specific text that appears on a button, or the size of a box (for example) will change according to the context. Windows is responsible for drawing these standard controls on the screen. The programmer simply describes the layout and dimensions of the visual elements, and Windows does the rest.[15]

The screen shot in Figure 3-4 also shows other, more sophisticated, visual elements: the scroll bars to the right and at the bottom of the document window, the toolbars containing the rows of buttons with a pictorial indication of the function of each, and the status line at the bottom of the screen. Add to these the standard menu bar and the application title bar, and you have examples of most of the visual elements in a Windows 3.1 program.

From the operating system's point of view, every single one of the interface's visual elements is a window. Not just the larger areas surrounded by the framing borders as in Figure 3-4, but virtually every visual element of the Windows interface, is a separately identified window. The operating system keeps track of all of the windows, and user actions performed in one window—for example, a mouse click on a checkbox—ultimately result in the system's sending a *message* to the application that owns the window. The message to the application takes the form of data that informs the application in which window the action took place and what happened in the window. Very often the application relies upon the system's default processing to take care of any action required in response to the message. For example, Windows itself will draw or remove the mark in a checkbox if the user clicks on the checkbox. Thus, a large amount of the code in Windows is devoted to handling all of these default actions, and individual application programs don't have to include equivalent functions. One of Microsoft's guiding principles in the design of Windows has been to include within the operating system functions that a majority of users or applications

15. Because Windows is responsible for drawing the controls, your Windows 3.1 applications will have the Windows 95 visual appearance when you run them under Windows 95. Since it is the system that displays the standard visual elements, a 3.1 application will take on the new look without any modifications.

will need. It's no surprise then when new visual elements such as an application toolbar—and the associated default processing—eventually appear in an operating system release. That's exactly what happens with Windows 95.[16]

The concept of window ownership is another notion central to the Windows system. Windows implements a strict hierarchy of windows. Every window must have a *parent window,* and any application may create, perhaps many, *child windows.* A child inherits many aspects of the parent, such as its default behavior. The hierarchical relationship also defines how window messages pass through the system: the youngest child window gets the first chance to process a message aimed at the window, and if it ignores the message, its immediate parent inherits the message. Ultimately the message may pass all the way to the top of the hierarchy so that the system itself can respond with the default message handlers.

The windows within our dialog box example are all child windows of the dialog box window. When the parent window disappears, so do all the child windows. When an application terminates, all of the descendant windows created by the application disappear (are "destroyed," in application programmer's parlance).

The programmer's term *control* actually refers to standard elements in the Windows interface that populate components such as dialog boxes and message boxes. Typically a control has some changeable data associated with it and will constrain what the user can do to the data. A checkbox, for example, allows only an on or an off condition, and a list box may allow the user to select only from a predetermined list of entries. The concept of a control is a little broader than this simple description indicates, but most applications use these kinds of controls. For application programmers, Windows makes the use of controls very easy by providing all the software to create, manage, and modify them and, subsequently, to determine user input.

Windows 95 User Interface Design

When contemplating changes to the appearance of Windows, the designer faces more considerations than the visual appearance of a particular element, considerations such as those itemized on the next page.

16. In accord with the same principle, network support and disk compression support have ultimately been incorporated into operating systems. Support for spreadsheet operations hasn't been and most likely never will be.

- What is the default behavior for a new window? Is it similar enough to an existing window type that applications can take advantage of common processing by the system?
- What behavior does a new window's appearance imply? A checkbox-like window that requires the user to enter a single letter or number will probably confuse most users, for instance.
- Is the new element useful for many applications and not simply for a single special case?
- Does the proposed new element or new appearance or behavior of an existing element actually help the user? That is, does the new or changed element provide an easier or more obvious way to do something?

Add these considerations to the more practical ones of large scale software development—how much memory is needed, how fast it will run, whether it can be finished in time—and you can see that changing the appearance of Windows 3.1 was more than just a facelift operation. The changes in the interface from Windows 3.1 to Windows 95 do aim to correct a number of flaws. But more impressive, a number of new user interface concepts make their first appearance with Windows 95. These ideas form the basis for the design of many of the new visual elements and for the design of the Windows 95 shell itself. In Chapter Five, we'll identify the problems in Windows 3.1 that Windows 95 aims to correct and look at the conceptual basis for the new appearance.

Windows Programming Basics

This book isn't about to try to teach you how to program for Windows. That subject has been explored comprehensively in hundreds of books and magazine articles over the last few years.[17] However, just to make sure that we embark on this voyage of discovery on an equal footing, let's review some basic information.

Event Driven Programming

Windows uses an *event driven* programming model that's almost more commonplace now than the procedural model everyone learned in

17. As ever, Charles Petzold's book *Programming Windows*, 3d ed. (Microsoft Press, 1992), remains the best introductory text.

school. First popularized by the Apple Macintosh operating system, event driven programming relies on external events to stimulate responses from an application. Mouse clicks and key depressions are the two most common external stimuli for a Windows application, although it's possible to translate any change in the application's environment into an event suitable for consumption by an application.

Windows feeds an event to an application in the form of a *message* that describes the change in the application's environment. Some messages are universal, such as those informing an application that the user has clicked on an application menu item. Other messages—for example, those indicating movement of the mouse cursor within an application window—are often of interest only to a particular type of application. Every message is associated with a specific application window, and each window has a *window procedure* associated with it. A Windows application receives messages by means of the *GetMessage()* API function, and calls Windows by means of the *DispatchMessage()* API function. Then Windows itself calls the appropriate window procedure, passing it the message to be processed. All messages are processed from within a queue that's maintained by the system and that preserves the order of the messages. If mouse click and keyboard entry messages, for example, weren't received and processed in the same order as the user entered them, the system would be out of control.

Message Handling

It used to be that every Windows application included the code fragments shown in Figure 3-5 on the next page—although you should notice one innovation in the code shown there. If you've written Windows programs, you probably have something very similar in your earlier programs. Windows applications rely upon the system to provide significant amounts of default processing. If an application isn't interested in a particular message, it simply ignores it and allows the system to apply its default response behavior to the message. Often the default processing means discarding the message altogether, and often it means that the window procedure for a particular message is simply not part of the application. For example, it is quite rare for an application to register a window procedure to handle messages sent to controls—the system's default handling of such messages is usually adequate.

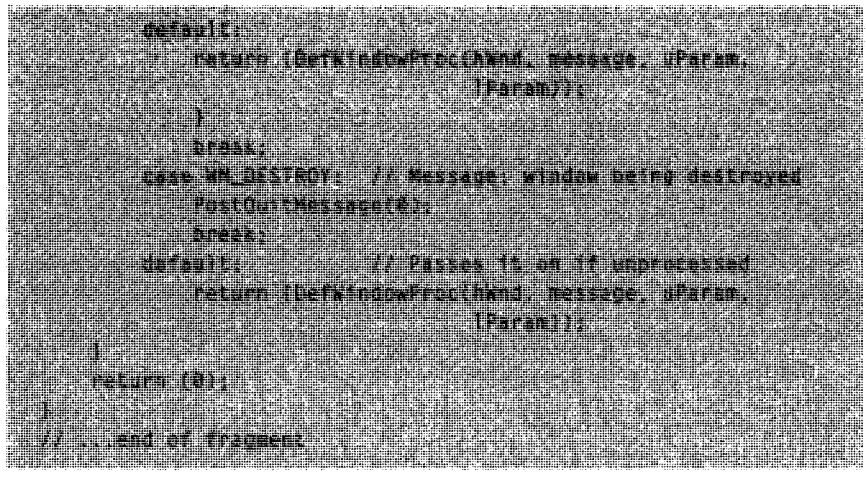
```
// Start of fragment...
// Acquire and dispatch messages until a WM_QUIT message is
// received.
while (GetMessage(&msg,          // Message structure
                  NULL,          // Handle of window receiving
                                 //  the message
                  0,             // Lowest message to examine
                  0)){           // Highest message to examine
    if (!TranslateAccelerator (msg.hwnd, hAccelTable, &msg)) {
        TranslateMessage(&msg); // Translates virtual key
                                //  codes
        DispatchMessage(&msg);  // Dispatches message to
                                //  window
    }
}
// ...end of fragment

// Start of fragment...
switch (message) {
    case WM_COMMAND:  // Messages: command from application
                      //  menu
        #if defined (_WIN32)
            wmId    = LOWORD(uParam);
            wmEvent = HIWORD(uParam);
        #else
            wmId    = uParam;
            wmEvent = HIWORD(lParam);
        #endif
        switch (wmId) {
            case IDM_ABOUT:
                lpProcAbout = MakeProcInstance((FARPROC)About,
                                               hInst);
                DialogBox(hInst,                // Current instance
                    "AboutBox",                 // Dlg resource to use
                    hWnd,                       // Parent handle
                    (DLGPROC)lpProcAbout);      // About() instance
                                                //  address
                FreeProcInstance(lpProcAbout);
                break;
            case IDM_EXIT:
                DestroyWindow (hWnd);
                break;
```

Figure 3-5. *(continued)*

Fragments of the Windows message loop.

Figure 3-5. *continued*



```
        default:
            return (DefWindowProc(hWnd, message, wParam,
                                  lParam));
        }
        break;
    case WM_DESTROY:  // Message: window being destroyed
        PostQuitMessage(0);
        break;
    default:          // Passes it on if unprocessed
        return (DefWindowProc(hWnd, message, wParam,
                              lParam));
    }
  return (0);
}
// ...end of fragment
```

Program Resources

Another common aspect of Windows programs is their use of identifiers called *handles* to reference every object within their environments: windows, memory blocks, files, communications devices, cursors, bitmaps, and so on. Handles are simply convenient numeric identifiers for resources that the system has allocated to a Windows program. Almost every Windows API function deals with a handle in one way or another. Sometimes a handle can be translated into a more direct reference—a memory address, for example. However, it's bad practice to do that, and under Windows 95 the unwritten rules for such translations have changed anyway.

Windows 95 Programming

Under Windows 95, the fundamentals of Windows programming haven't changed. The event driven model is still the basis for how you write a Windows program. However, there are some evolutionary changes in writing a program for Windows 95:

- Microsoft is all but forcing developers to move to Win32 as the preferred Windows API. There are a lot of good technical reasons to go to 32-bit programs anyway, but the fact that the new capabilities of Windows 95 are accessible only to Win32 applications tends to predetermine the result.

- The programmer's access to the new capabilities of Windows 95, notably 32-bit programs and preemptive scheduling, will introduce new twists in the already complex Windows programming model. If you don't already know how to develop applications for a preemptive multitasking system, Windows 95 forces you to learn. There are also some subtle changes that the 32-bit API engenders in application code—if you looked at the code in Figure 3-5, you saw one example.
- Microsoft's Object Linking and Embedding (OLE) technology represents a massive investment in a new programming methodology that may well transform Windows programming and the nature of Windows applications. OLE has been available in advance of the Windows 95 release, but its presence as a standard component of the Windows 95 product is likely to ensure that a lot of programmers will spend a lot of time learning it.
- The programming tools now available for Windows stress more and more the object-oriented programming model evident in languages such as C++. Windows is by its nature an object-oriented environment, although purists can point to areas in which Windows deviates from a pure object-oriented model. The new tools for Windows programming tend to hide these minor deviations, and with the emphasis that Microsoft now places on OLE and the future promise of Cairo, object-oriented programming is likely to be the discipline in vogue for the next few years.

Although everything you worked hard to learn about Windows programming is still valid, there are some new aspects that Windows 95 will tend to bring into focus. OLE is not the least of these and is by some estimates as complex as the entire Windows 2.0 product ever was. However, if you're comfortable with the basic concepts of events, messages, message queues, window procedures, handles, and windows, you shouldn't find anything in the following chapters to be incomprehensible.

Conclusion

In this chapter, we took a tour through a lot of the basic terminology and some of the inner workings of Windows. If you knew most of this Windows lore already, you're ready for the new acronyms and some of the architectural changes introduced with Windows 95. If you didn't know your way around Windows, I hope you're ready for a second heavy dose.

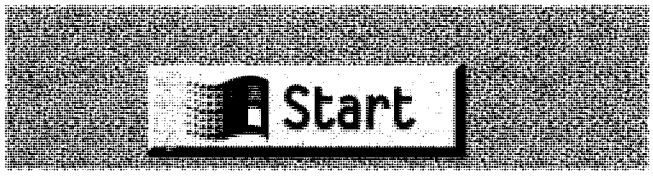
We looked at several of the new features of Windows 95 in this chapter but ignored a lot of the detail. Chapter Four is where we're cleared for the approach to Chicago.

References

Duncan, Ray, et al. *Extending DOS.* 2d ed. Reading, Mass.: Addison-Wesley, 1991. A collection of lengthy papers about different aspects of squeezing more memory and more function from MS-DOS. The book includes a good discussion of DOS extenders and the DPMI specification.

Intel Corporation. *MS-DOS Protected Mode Interface Specification.* The definitive specification for version 0.9 of DPMI. There's also a version 1.0, but since Windows itself supports only version 0.9, this is the de facto standard. To get a free paper copy, call Intel at 1-800-548-4725.

Petzold, Charles. *Programming Windows 3.1.* 3d ed. Redmond, Wash.: Microsoft Press, 1992. A classic in its own way. The best introduction to Windows programming there is. If we're lucky, Charles is hard at work on the Windows 95 version.



CHAPTER FOUR

THE BASE SYSTEM

In this chapter and the next, we'll examine the two features of Windows 95 that most differentiate it from its predecessors. Of all the new features in Windows 95, the most prominent to the user will be the new appearance and the new system shell—the most obvious changes from Windows 3.1—and that's what we'll look at in Chapter Five. For the programmer, the support for a native 32-bit API will probably be the most closely studied new feature in Windows 95. But the 32-bit API is merely the best-documented manifestation of the changes in the underlying operating system. In Windows 95, Windows finally becomes a complete operating system. No longer is it simply a "graphical DOS extender," some critics' characterization of the earlier versions of Windows. In Windows 95, many new or revised components now make full use of the 32-bit protected mode of the 386 processor. The operating system within Windows 95 is the subject of this chapter.

Simply looking at the feature highlight list for the base operating system gives you an indication of how much is new and how much work has gone into this part of Windows 95:

- For all intents and purposes, real mode MS-DOS is gone. Finally Windows is a complete operating system with no reliance on MS-DOS and its real mode architecture and limitations.
- A new filesystem architecture and 32-bit protected mode implementation of the FAT filesystem eliminate the last major dependency of Windows on MS-DOS. The new filesystem also provides significant system performance improvements.

- Windows 95 provides full support for 32-bit applications, including a 32-bit Windows API and protected, private address spaces.
- Windows 95 provides for the preemptive scheduling of Windows applications.
- Windows 95 provides architected support for multiple simultaneous network connections.

Naturally, whatever changed in Windows 95 had nevertheless to remain compatible with Windows 3.1 and MS-DOS. The developers had the ever present specter of compatibility looking over their shoulders.

And the designers of Windows 95 had to recognize Windows NT as a preexisting operating system in much of their work. Sometimes the obligation to Windows NT helped. Windows 95 picked up components of the disk device driver architecture used in Windows NT, for example. And sometimes deference to the earlier Windows NT created quandaries: which subset of the Windows NT API set Windows 95 should fully support, for instance. As we examine the system's features, we'll draw a number of comparisons between Windows 95 and Windows NT.

What we'll concentrate on in this chapter are the underlying architecture and the major functional components of the operating system. While the project was under development, the Windows 95 team publicly referred to this collection of software as the *base system,* or simply the *base OS.*[1] Throughout the project, there was a lot of internal and external discussion and speculation about a protected mode MS-DOS version 7.0 that would provide the operating system functionality required by Windows 95. By and large, this version of MS-DOS (if it appears) will be the operating system components of Windows 95 in a different package. Since we're concerned with Windows only, we won't go into what might or might not appear in MS-DOS version 7.0.

Windows 95 Diagrammed

Software designers often discuss an operating system as if it were a living, breathing entity. Reducing such an organism to simple diagrams can't provide a complete picture of either its complexity or the subtle interactions among its different components. But given our medium,

1. Microsoft code-named the OS components Jaguar and Cougar. There were also dragons stalking the halls. Interesting place to work.

diagrams are what we have.[2] Figure 4-1, a variation on Figure 3-1, provides just such an inadequate view of the system's most important components.

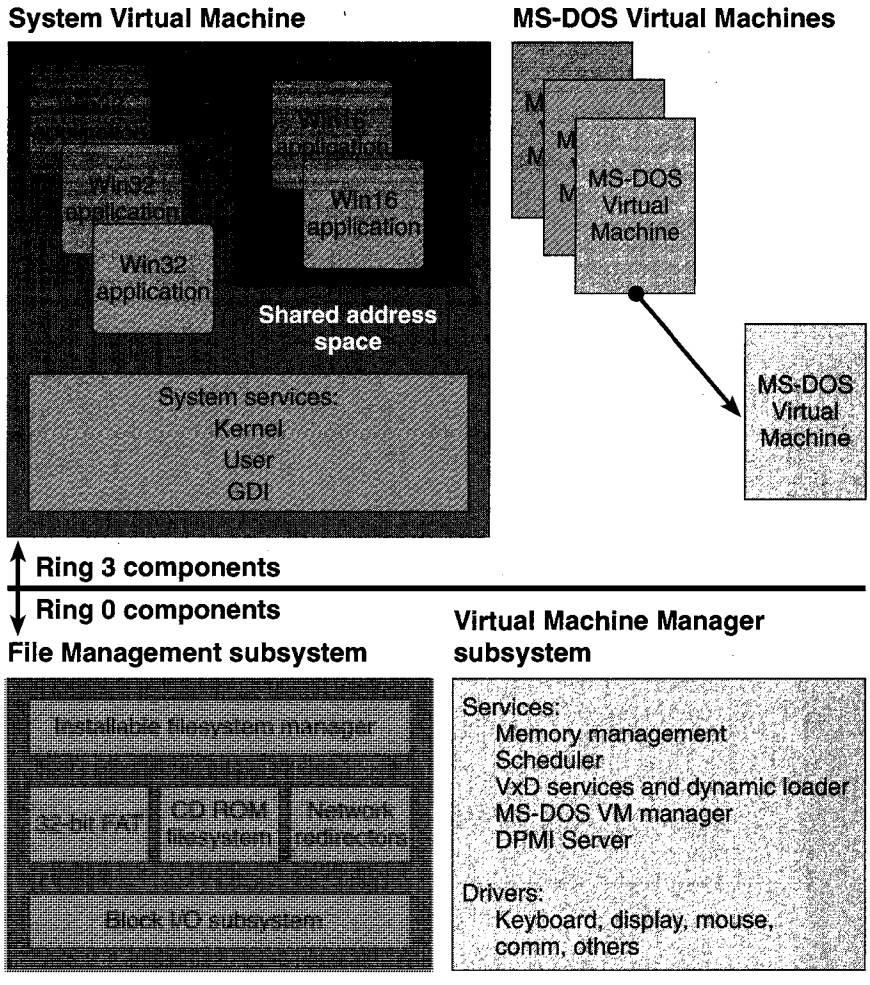


Figure 4-1.
Windows 95 system architecture.

2. One Microsoft designer maintains that drawing a block diagram of Windows NT gives you a neat, concise presentation showing how the system really does work. For Windows 95, a similar representation is a little more chaotic, but the diagrammatic oddities usually point to important concerns—namely, compatibility and performance.

It would be difficult to point to a single box as the base operating system since aspects of the low-level design permeate Windows 95. In this chapter, we'll concentrate on the functions provided by the Virtual Machine Manager and on some details of the System Virtual Machine architecture:

- Scheduling and memory management services
- The management of Windows-based applications within the System Virtual Machine
- The management of the MS-DOS virtual machines
- The foundation for the Windows API layer

We won't get into all of the extremely low level details of how these pieces work. We'll look at the architecture and at some of the more interesting implementation details.[3] Needless to say, you should be familiar with the material presented in Chapters Two and Three before diving into this chapter.

Windows 95 Surveyed

Let's first take another brief tour through the system and review the important components. Many aspects of the Windows 95 design are similar to aspects of the design of Windows 3.1 that you already know about. In particular:

System Virtual Machine. Windows applications all run within the context of the system VM. The 16-bit applications (the "old" Windows applications) share a single address space. The new 32-bit support provides each new application with a private address space.

MS-DOS Virtual Machines. Windows 95 supports the execution of multiple MS-DOS programs running in either virtual 8086 mode or protected mode.

Virtual Machine Manager. The VMM is the real heart of the operating system. It provides low-level memory management and scheduling services as well as services for the virtual device drivers.

3. No doubt there will be other books that do take on the Herculean task of looking at all of the details. The "References" section at the end of this chapter lists a few of the books that covered the details for Windows 3.1.

The major new component of Windows 95 is the File Management System. It's a completely redesigned subsystem that supports multiple concurrently accessible filesystems. Barring any old MS-DOS device drivers that might be present to support a particular device, the entire File Management System is protected mode 32-bit code. Its design supports local disks and CD ROM devices as well as one or more network interfaces by means of an *installable filesystem interface* (*IFS* for short). If you're really well connected, you can hook up and use your hard disks, your floppy disks, your CD ROM, your Bernoulli box, your Windows NT server, and your NetWare network and never leave protected mode the whole day. In Windows 3.1, it was MS-DOS that provided the filesystem support for local disks. Support for CD ROM devices and network filesystems was, at the very least, confused and confusing.[4]

The system services called upon by Windows applications—for graphics, window management, and the like—are all still there, and they retain the Kernel, User, and GDI names they had in previous versions of Windows. The major change in the system services subsystem is its support for 32-bit applications. Apart from their different memory management requirements, 32-bit applications use a full 32-bit Windows API and call upon services that are now implemented using 32-bit code. Making the mixture of 16-bit and 32-bit components cooperate effectively and with good performance was one of the major design and implementation challenges the Windows 95 team faced.

Protection Rings in Windows 95

Windows 95 exploits the Intel 386 processor's ability to support multiple privilege levels. Since the handling of these rings of protection tends to affect several aspects of system design, it's worth reviewing their use up front. Windows 95 runs the processor using privilege levels zero and three. The ring zero components are what you normally think of as the operating system proper, including, for example, the lowest levels of memory management support. Ring zero software has omnipotent power over the system: all the processor instructions are valid, and the software has access to critical data structures such as the page tables. Clearly, it behooved the system designers to ensure that the software running at ring zero would have a very good reason to be there and be completely reliable. For the most part, Windows 95 ensures these

4. The first release of Windows for Workgroups improved this situation some, and version 3.11 made it better yet. The protected mode FAT filesystem made its debut in the 3.11 release of Windows.

conditions. The lapse is in the facility that allows the user to install one or more new virtual device drivers to support an add-on hardware device or provide some systemwide software service. VxDs always run at ring zero, and if one of them fails, it can cripple the entire system. Unfortunately, the performance overhead that would have been incurred by putting each VxD in a private address space so that failed drivers could be isolated and halted was deemed unacceptable.[5]

Windows applications and MS-DOS applications always run at ring three, so their privileges are significantly restricted. Also running at ring three are the central components of the Windows graphical environment: Kernel, User, and GDI. The term *Kernel* has been so prevalent in descriptions of how earlier versions of Windows operate that we'll keep its sense in that context rather than adopt the more classic use of the word to describe the ring zero components of Windows 95.

Some operating systems try to use the other privilege levels offered by the Intel 386 processor. Windows 95 isn't one of them. The two-ring model (sometimes called "kernel and user modes") works pretty well for most needs. The Windows 95 designers could have come up with ways of using the other rings—running user installed VxDs at ring one to reduce the system integrity problem, for instance. But this line of thinking leads rapidly to a consideration of the various tradeoffs, notably implementation effort and system performance vs. real user benefit. A ring transition on the Intel 386—a change of control from one processor privilege level to a different one—is expensive in terms of execution time.[6] A lot of processor controlled validation and register reloading occurs whenever there's an alteration in processor privilege level—that is, a jump between rings—so minimizing such transitions represents a big benefit to system performance. This is also why most of the code for the Windows graphical system runs at ring three. Incurring a ring transition for every Windows API call would likely result in system performance reminiscent of Windows 1.01 running on an IBM PC XT.

Windows 95 Memory Map

The 386 provides a 4-GB virtual address space, and Windows 95 uses it all. Within this virtual address space, the different system components

5. The problem did get quite a bit of attention. The Windows 95 development tools do include new VxD debugging and parameter validation capabilities.

6. A direct subroutine call to code in another segment takes 20 clock cycles on the 486. If a ring transition is involved, you need to budget 69 clock cycles. And that's one way only. The return path is expensive too.

and applications occupy regions with fixed boundaries. Figure 4-2 shows the basic memory map for the system. One duty of the Virtual Machine Manager (VMM) is to map this 4-GB virtual address space into the available physical memory.[7]

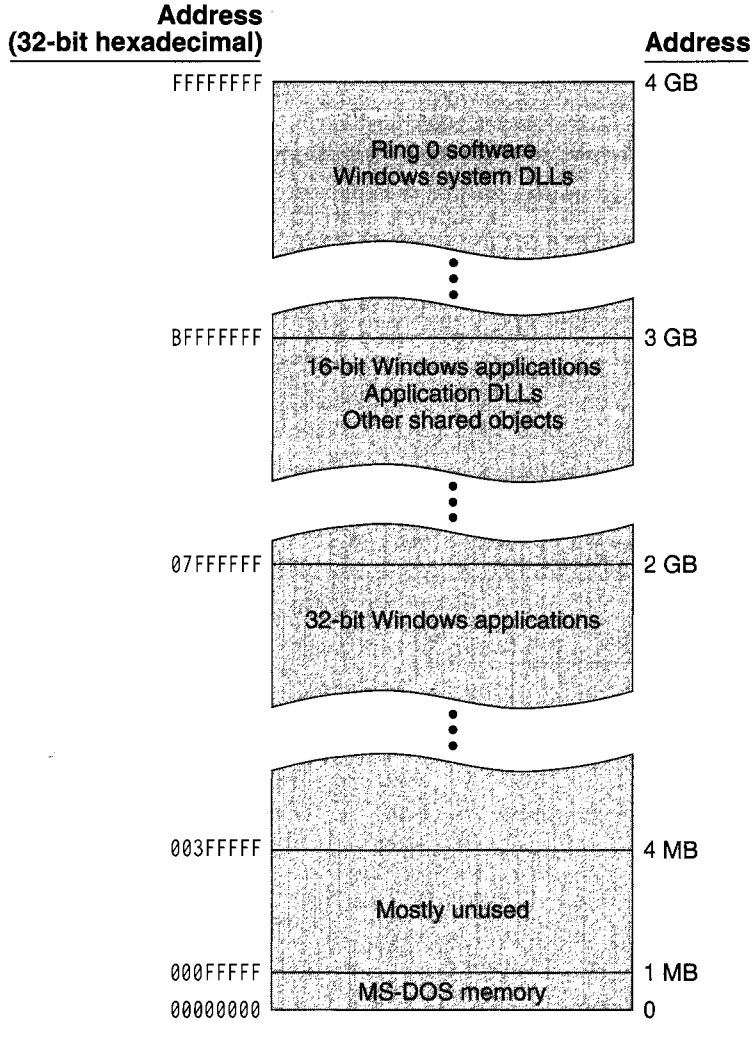


Figure 4-2.
Windows 95 system memory map.

7. The Windows 95 base operating system uses two selectors—28 and 30—for code and data. The base and limit for the associated descriptors are set at 0 and 4 GB, effectively providing access to the entire virtual address space.

In the system memory map, the lowest 1 MB of the virtual address space is used for the currently executing MS-DOS VM. Each VM also has a valid memory map within the 2-GB to 3-GB region.[8] This mapping allows the system itself to address the memory of a VM regardless of whether it is active. But when an MS-DOS VM runs, it's also mapped to the bottom 1 MB.

Within the virtual address space of a 32-bit Windows application, the standard development tools use 4 MB as the default load address. You can choose a lower address, but you'll incur a lot of overhead with all the fixups the system will have to carry out when it loads the application. Loading into the 4-MB to 2-GB region is immediate. The 4-MB application load address matches the address Windows NT used for loading 32-bit applications in its first production release, so it's a sensible choice. The lowest 16K of each 32-bit application's address space (that is, virtual addresses 0 through 3FFF) is invalid. This deliberate design decision aims to trap program errors. One of the most common programming errors is the erroneous use of a null program pointer. Under Windows 95, the 0 address will generate a memory fault, an error likely to be caught by the developer and not get as far as the user.

Tasks and Processes

One significant change in Windows 95 that needs to be appreciated at the outset is the change in terminology from *task* to *process*. Windows 3.1 documentation usually used the word *task* to describe the running instance of a program. Windows 95 aligns itself with Windows NT in using the word *process* to describe the same thing. A lot of the Windows 3.1 documentation wasn't particularly rigorous in using the word *task*, so you can actually find both words used. In Windows 95, the word *process* refers, at least in the case of 16-bit Windows applications and MS-DOS applications, to the "task" you already know about.

If you study the documentation for Windows 95, you'll see that API calls such as *GetCurrentTask()* are marked "deleted" or "obsolete," and you're referred to the new API. (Yes, you'll find *GetCurrentProcess()* instead.) Of course, the compatibility constraints that govern Windows 95 mean that the system must still support the older task API calls, so they aren't really "deleted" in the true sense of the word. Even though

8. The 2-GB low address boundary of the shared memory region moved from 1 GB to 2 GB in successive test releases of Windows 95. (Although such a move wasn't contemplated, it may even have moved again by the time you read this.) It isn't an address you should depend on for any reason.

Microsoft doesn't expect anyone to develop new 16-bit applications, you could still do that, and the task APIs would be available to you. Once you enter the 32-bit world, though, a "process" is what you have and the process APIs are what you use.

Virtual Machine Management

The virtual machine concept that was so important in the very first implementation of Windows on the Intel 386 is alive and well in Windows 95. The Virtual Machine Manager is truly the heart of the Windows 95 base system. The efficiency of the VMM has a major impact on the performance of the whole system, and some of the most complex components of the OS live there. The code for the VMM consumed some of the best efforts of the development team, and they've added a lot of new functionality:

- 32-bit Windows applications are preemptively scheduled within per-process private address spaces.
- Many new system primitives related to the preemptive scheduling environment are available to VxDs.
- VxDs can be dynamically loaded and paged, which reduces the working set for the system.

Also, within the Windows User module, each 32-bit application obtains a private message queue—eliminating the possibility of a single application's locking up the entire system, which can happen in Windows 3.1.

Windows 95 uses the same two basic types of virtual machine that Windows 3.1 did:

- The system VM, in which the Windows Kernel, User, and GDI components as well as all the Windows applications run
- The MS-DOS VMs that run a single MS-DOS session each, with applications running in either virtual 8086 mode or protected mode

Real MS-DOS

Despite earlier statements to the contrary, MS-DOS is still alive and well in Windows 95. (You didn't really think it had gone away, did you?) The

code and data for the current release of MS-DOS (version 6.22) will be present on the Windows 95 disks at shipment, although it's not clear exactly how the packaging and pricing issues will be resolved. Here's why MS-DOS is still around:

- Windows 95 supports a *single MS-DOS–based application mode*—to give it its official title. This mode is for MS-DOS applications that can't run under Windows—typically, game programs that have stringent timer control requirements.
- The software in the "hidden" VM, where Windows sets up the global MS-DOS context for all other VMs, has to come from somewhere. MS-DOS itself is the obvious candidate for providing the MS-DOS context.

Earlier in the development of Windows 95, the intention was to use MS-DOS as the bootstrap loader for the system. Rather than reinvent the code that brings the system to life, processes the CONFIG.SYS and AUTOEXEC.BAT files, and then runs Windows proper, Microsoft planned simply to use MS-DOS. Eventually, the boot process was put into the WINBOOT.SYS module. The module contains a lot of MS-DOS code, but it's tailored to the job of getting Windows 95 into memory and starting it.

The big difference in Windows 95's relationship with MS-DOS is that if you run only Windows applications, you'll never execute any MS-DOS code. As successive versions of Windows have appeared, each has supported more and more of the MS-DOS INT-based software services, and Windows applications have had an ever decreasing need to switch in and out of virtual 8086 mode to execute MS-DOS code. The big exception to this (up to Windows for Workgroups version 3.11) has been support for the filesystem services. Windows 95 finally breaks all ties with the real mode MS-DOS code, and with few exceptions even the existing 16-bit Windows applications follow a protected mode path through the new File Management System to the disk and back.

Virtual Machine Scheduling

Process scheduling in Windows 95 is so closely tied to the management of virtual machines that it's appropriate to examine scheduling as part of the VMM discussion. The Windows 95 scheduling algorithms deal with

virtual machines, processes, timeslices, and priorities similarly to the way Windows 3.1 did. Windows 95 also introduces *threads,* the principal objects that the system scheduler deals with. The thread is now the basic unit of scheduling in Windows 95. If you're familiar with Windows NT or OS/2, you're accustomed to dealing with threads. A thread

- Is an execution path within a process.
- Can be created by any 32-bit Windows application or VxD running on Windows 95.
- Has its own private stack storage and execution context (notably processor registers).
- Shares the memory allocated to the parent process.
- Can be one of many concurrent threads created by a single process.

Threads are sometimes called "lightweight processes" because creating and managing them are relatively simple operations. In particular, the fact that threads share all the code and global data of the parent process means that setting up a new thread involves only minimal amounts of memory allocation. When Windows 95 loads an application and creates the associated process data structures, the system sets up the process as a single thread. Many applications will use only a single thread throughout their execution lifetimes. But an application can (and many do) use another thread to carry out some short term background operation. Under Windows 3.1, waiting for a word processor to load a large document can be tedious. If you change your mind halfway through, you still have to sit and watch the hourglass cursor for a while before you can do anything else. Under Windows 95, the application can create one thread to load the document and another to manage a dialog with a Cancel button. Any time you want to, you can interrupt the document loading operation with a single click.

Thread services are available only to 32-bit applications and VxDs under Windows 95. MS-DOS VMs and the older 16-bit Windows applications can't call the thread APIs. An MS-DOS VM represents a single thread: in simple terms, an MS-DOS VM is a process is a thread. Every 16-bit Windows application uses a single thread of execution, and the cooperative multitasking model for older Windows applications is preserved. Any 32-bit Windows application or VxD can create additional

threads, and Windows 95 can schedule all these threads preemptively—adding a whole new facet to Windows multitasking.[9]

The Windows 95 Schedulers

There are two schedulers within the Windows 95 VMM: the *primary scheduler,* which is responsible for calculating thread priorities, and the *timeslice scheduler,* which is responsible for calculating the allocation of timeslices. Ultimately, the timeslice scheduler decides what percentage of the available processor time to allocate to different threads. If a thread doesn't receive execution time, it's *suspended* and can't run until the schedulers reevaluate the situation.

Here's how the scheduling process works:

1. The primary scheduler examines every thread in the system and calculates an *execution priority* value for the thread, an integer between 0 and 31.[10]
2. The primary scheduler suspends any thread with an execution priority value lower than the highest value. (The highest value doesn't necessarily mean the value 31. If two threads have the execution priority value 20 and every other thread has a priority value lower than 20, then 20 is the highest value until the next priority recalculation.) Once a thread is suspended, the primary scheduler pays no further attention to the thread as far as priority calculation during this timeslice is concerned.
3. The timeslice scheduler then calculates the percentage of the timeslice to allocate to each thread using these priority values and knowledge of the VM's current status.
4. The threads run. By default, the primary scheduler will reevaluate the priorities every 20 milliseconds.

In the example in Figure 4-3, two of the five active threads (B and D) have execution priority values of 20 and the other three threads

9. Although these threads can correspond to radically different program types, the system represents each thread using the same data structure. Thus, the scheduler, along with other 32-bit system code that uses these internal data structures, could be implemented without the team's having to worry about 16-bit to 32-bit translation idiosyncrasies.

10. That this is the same priority model as Windows NT's reflects a design guideline for Windows 95: "where it makes sense to, be the same as Windows NT."

have lower priorities. The timeslice scheduler will therefore divide the next timeslice between threads B and D.

Three control flags maintained for each VM also play into this process. *VMStat_Exclusive* tells the scheduler that the VM in question must receive 100 percent of the next timeslice; neither of the remaining two flags is set. One of the remaining two flags—*VMStat_Background* and *VMStat_High_Pri_Background*—must be set if the scheduler is to

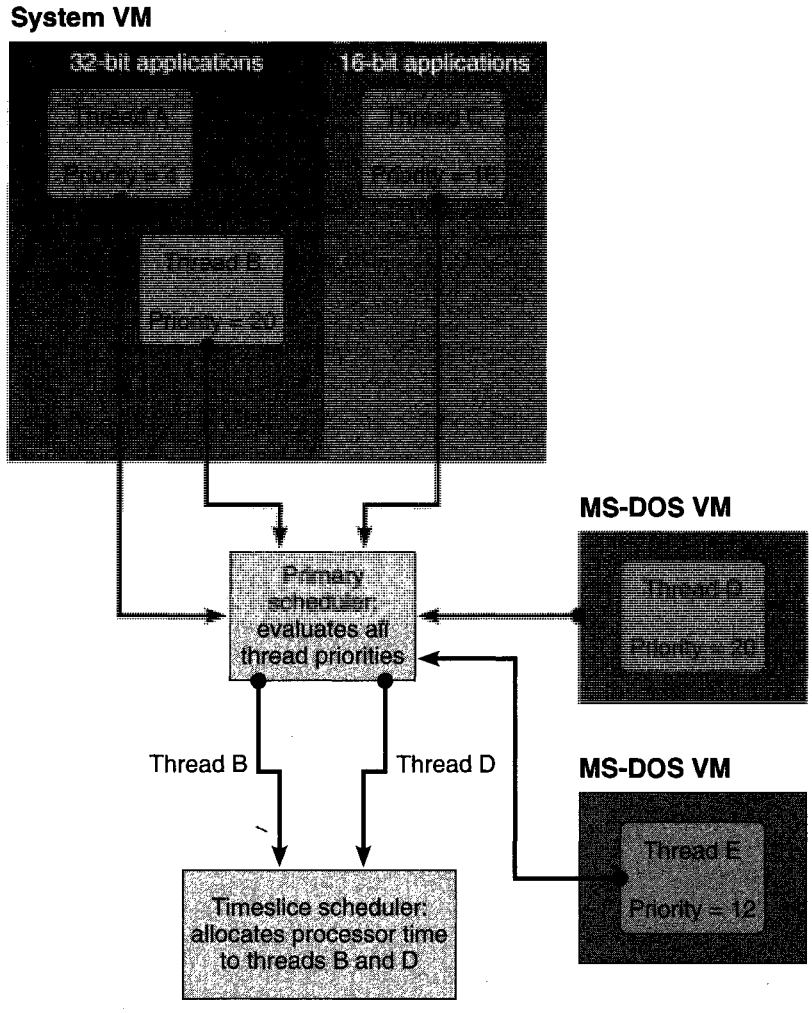


Figure 4-3.
Windows 95 thread scheduling.

grant a background VM any allocation of the next timeslice; otherwise, the foreground VM gets the entire allocation.

Scheduling Within the System Virtual Machine

All the Windows application threads run within the System VM context. The System VM is the only VM that supports multiple threads: one for each 16-bit application and at least one for each 32-bit Windows application. As you can see from the discussion of the scheduling algorithm, it's possible (in fact probable) that the System VM will frequently contain multiple nonidle threads with equal high priorities.

To handle this situation, the timeslice scheduler adopts a *round robin* scheduling policy to ensure a fair allocation of execution time among threads of equal priority. Once a thread within the System VM consumes its allocated execution time, the scheduler puts it at the end of a queue of threads with equal priority. This classic technique ensures that each thread at the highest priority level has an equal opportunity to consume processor time. If the chosen thread fails to consume all of its allocated processor time, the scheduler hands the processor to the next thread of equal priority in the System VM and allows it to use the remainder of the timeslice.

Controlling the Scheduler

Two different influences control the scheduler. One is its own internal algorithms that try to provide a smooth multitasking environment with each thread receiving an equitable share of processor time. "Smooth" in this context is really a user perception—the goal is to provide a thread with enough processor time to get work done but not so much time that other threads are locked out for long periods. Erring on the side of providing too much processor time to a thread will give the user an impression of slow response as he or she waits until the system switches to the new thread. Providing too little processor time to threads will give the user an impression of jerky response as the system switches among threads. The other influence on the scheduler is the direct calls on system services that VxDs might make.

Internally, the scheduler uses three techniques to help it meet its goal of equitable distribution of processor time for an impression of speedy and smooth response:

Dynamic priority boosting allows the primary scheduler to briefly raise or lower the priority of a thread. For example, a keystroke or a mouse click indicates that the receiving thread's priority should be boosted.

Timed decay causes the boosted priority of a thread to gradually return to its usual value.

Priority inheritance rapidly turns a low-priority thread into a higher-priority thread. Typically, a thread's priority is inverted to allow a low-priority thread to rapidly complete its use of an exclusive resource that high-priority threads are waiting for.[11]

The VMM includes a large number of services available to VxDs. The operating system uses these services extensively to control multitasking operations. For software authors brave enough to dive in, the multitasking services are all available from within user installable VxDs. These services allow a VxD to inquire about current scheduling conditions—priorities, timeslices, VM focus, and other parameters—and to adjust those conditions.

Threads and UAEs

One of the problems facing the Windows designers has always been how best to deal with applications that fail during execution. Whether you call such a crash a UAE or a general protection fault, it comes from a bug—probably in the application itself, although the user tends to blame Windows. It's unlikely that any generation of Windows application designers will deliver totally bug-free software, so Windows itself has to be able to deal with application crashes. This involves two things:

- Handling the program failure gracefully—meaning allowing the user to close the application with a minimum of fuss and no lost data.
- Cleaning up afterwards. Apart from open files, the application undoubtedly owns handles to system resources such as memory segments, pens, and brushes. If the system can't free up the memory these resources occupy, the available free resources are reduced.

The most common application program error resulting in a crash is an addressing error. Typically, the bug causes the program to try to use an invalid pointer to some object. A 0 address is the most common case, which is why address 0 is always an invalid address for every Windows

11. Windows 95 immediately adjusts the inherited priority back to its normal value once the contention condition is past.

95 application. Such an addressing error causes a *general protection fault* on the 386, and eventually the user sees a dialog box that provides the name of the program module that caused the fault and the option to close the erring application. Of course, this information and the option to close the application don't help the user very much, and often the system behaves very strangely even after the user closes the application and dismisses the dialog box.

Windows 95 addresses this problem in two ways. First, the general protection fault handler runs as a separate thread within the system. Thus, rather than having the fault and the closing of the application handled from within the application context, which may by now be in a hopelessly messed-up state, Windows 95 has the fault dialog and program termination managed by a thread in a known (good) state.

The system has already tagged every allocated resource with a thread identifier, so if a thread terminates abnormally, the system can search its tables for any resources the thread owned and return them to an unused state. All global memory, window resources, logical brushes, device contexts, and other resources are available for reuse after this postmortem cleanup. The cleanup goes into immediate effect if a 32-bit thread fails. Amazingly, one of the "techniques" used by some existing Windows applications relies on allocated resources remaining available even after the application quits. For this reason, the resource cleanup can't take place until the system notices that there are no 16-bit applications running. Then any remaining allocated resources can be returned to the free pool.[12]

Threads and Idle Time

Another use of the thread mechanism is to schedule background activities that can run when the system is quiescent.[13] Waiting until the system is quiescent ensures that the maximum number of processor cycles remains available to applications.

12. This technique works also when an application simply "forgets" to release a resource, such as a display context, before exit.

13. In Windows 3.1, there was a background VxD that wrote modified memory pages out to the swap file. When no applications were running, this process woke up and ensured that the swap file images matched the memory images of the currently executing programs. Experiments showed that this really wasn't a big performance win, and the technique was dropped in Windows 95.

Application Message Queues

The event driven nature of all Windows applications calls for the system to provide an effective means of delivering messages to every application. A message is sent at the behest of a device driver (representing the occurrence of some external event such as a mouse click), by another Windows application, or by the system itself (for example, the system will notify other processes when a new application starts). The system puts all the hardware-initiated messages into a data structure called the *raw input queue.*

A classic problem with Windows 3.1 is that every Windows application draws messages from a single systemwide message queue. This message queue contains a processed form of the raw input messages suitable for application consumption as well as all the other messages that flow through the system. Whenever a process asks for a message (usually with a *GetMessage()* call), the system simply delivers the message at the head of the queue. Until the process yields control of the CPU, the system doesn't try to deliver any more messages. Since there is no preemption in Windows 3.1, if an application fails, the flow of messages—and consequently the system—comes to a halt. No doubt you've seen this phenomenon when an application puts up the hourglass cursor and goes to sleep—sometimes forever. Clicking the mouse on other windows doesn't help the situation in the least.

Unfortunately, even if Windows 3.1 were to provide a preemptive multitasking environment, a single message queue would still cause the same problem. For example, suppose that two messages (A and B) destined for the same process were at the head of the queue and that the process accepted message A and then failed, looping endlessly. The timeslice would expire, and the system would reschedule and grind to a halt—unable to deliver message B to the recalcitrant process.

To prevent this kind of situation, Windows 95 supports multiple message queues, a design improvement it shares with Windows NT. Since the efficient flow of messages is vital to good response times and smooth multitasking, this design technique is key. It ensures that a single errant application can't lock up the entire system.[14] The multiple queue technique is called *input desynchronization,* and Figure 4-4 on the next page shows how it works.

14. For the most part. There's still a design problem associated with the 16-bit application subsystem that we'll look at later in this chapter.

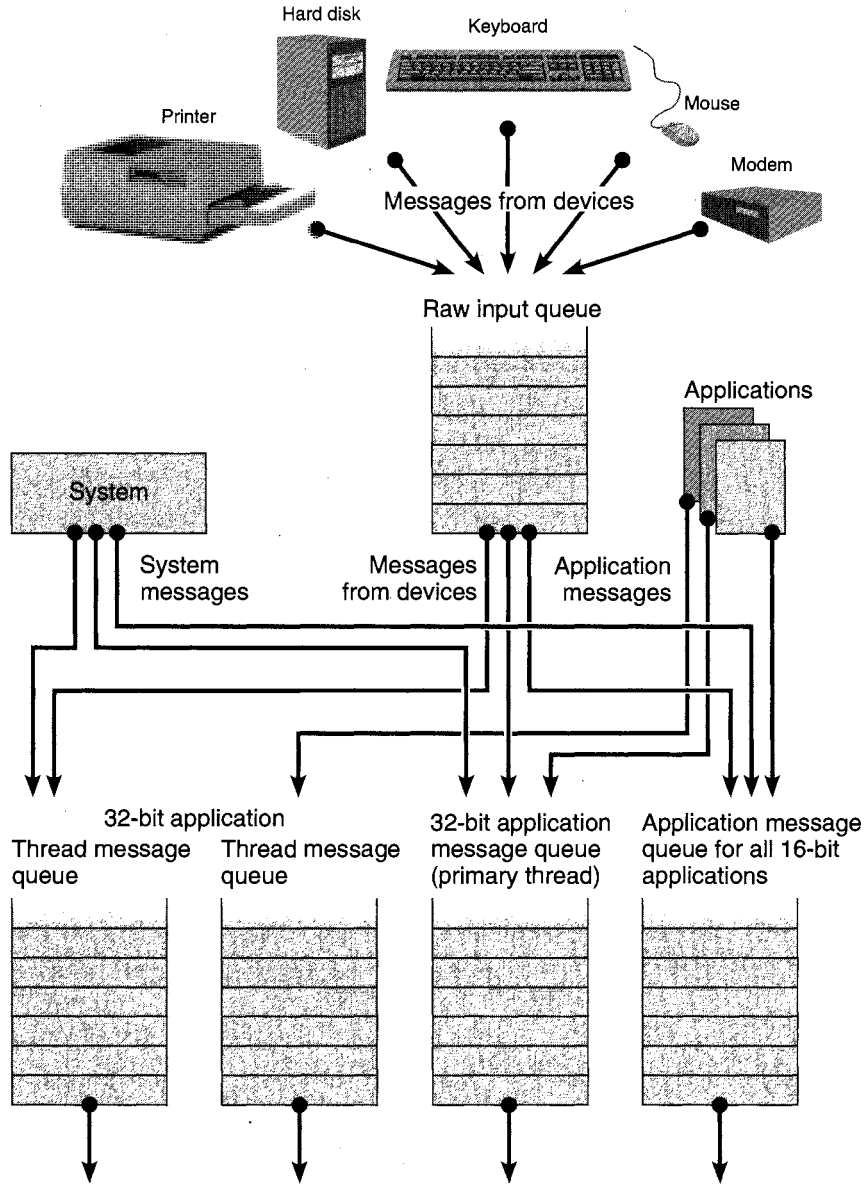


Figure 4-4.
Multiple queue message delivery under Windows 95.

Under Windows 95, new messages are put into the raw input queue only briefly. An execution thread within the system regularly empties this queue and moves the messages to one of these queues:

- A single queue for all 16-bit applications—meaning that the behavior for these applications is exactly as it was under Windows 3.1
- A private per-thread queue for 32-bit applications

Messages generated by the system itself or by other processes move straight to the private queues. There's a small amount of internal buffering if the system is extremely busy, but most of the time that isn't necessary. When a 32-bit process first runs, it has a single message queue associated with its primary thread. If the process creates another thread, the system doesn't immediately create another message queue, though. The system creates another message queue only when the second thread makes its first message queue–related call. If a thread doesn't need a message queue, the system doesn't waste any resources building one.

Physical Memory Management

Underlying the virtual machines and the virtual address space supported by Windows 95 are the confines of the physical memory present on the host system. Managing physical memory is the process of choosing which pages within the system's 4-GB virtual address space to map to physical memory at any instant in time. The system swaps the remaining active pages in the virtual address space to and from the hard disk, reassigning physical memory pages as it needs to. Many physical pages—for example, those occupied by the memory resident components of the kernel—have their use determined during system startup. These pages never change roles and don't figure in the memory management process. On a system with 4 MB of RAM and a small (probably very small) disk cache, you can expect roughly 1 MB of memory to be locked down this way. Several software components contend for the remaining physical memory: dynamically loaded system components, application code and data, and dynamically allocated regions such as DMA buffers and cache regions for the filesystem.

The Windows 95 physical memory manager is brand-new code. The main reason for rewriting the existing memory manager was the proliferation of memory types that Windows 95 has to deal with. Along with all the memory page types that Windows 3.1 has to manage, Windows 95

memory page types include 32-bit application code and data, dynamically loadable VxDs, memory mapped files, and a dynamic filesystem cache.[15] This increase in complexity was enough to dictate a rewrite.

Unlike the design of a multiuser system, in which the operating system has to worry about equitable sharing of the precious memory resource, the Windows 95 design allows you to fill your memory as you wish. All available physical memory pages are created equal, and both the system's dynamically loaded components and running application programs compete for available memory pages. You want an application to run as fast as possible, so the application is allowed to fill as much physical memory as can be made available. Over an extended period, machines with 8 MB or less of memory are likely to gradually fill all the available memory and have to start paging.[16] Note that the system imposes a restriction on the total amount of memory an application can lock—if this weren't controlled, it would be possible to reach a deadlock situation. Once physical memory is full, the next page allocation request starts the paging process. An interesting side effect of this design is that there is no reliable way for an application to determine how much memory is available in the system. The *GlobalMemoryStatus()* API reports various statistics about the system's memory, but the report is a snapshot of current conditions, and calling the API again will probably yield different results.

The paging algorithm in Windows 95 is a standard *least recently used* (*LRU*) technique that re-allocates the oldest resident pages when new requests must be satisfied.[17] Pages come and go from different places: most pages are either directly allocated in memory (as a result of a request for new data pages) or loaded initially from an application's .EXE file. Subsequently, these pages travel back and forth between physical memory and the swap file. The system always loads pure code

15. The VCache filesystem caching VxD interacts with the physical memory manager, claiming and releasing chunks of memory that can then be allocated to the individual filesystem drivers for cache usage.

16. Windows 95 remembers what it loaded, and even after an application exits, its code pages may remain in memory for some time. If the pages aren't taken for some other purpose and the user happens to run the same application again, the pages are still there and can be reused.

17. Early test releases of Windows 95 used a simple page-at-a-time paging algorithm. Late in 1993, the developers began experimenting with clumping pages together and paging a block of pages in each operation. At the time of this writing, page out operations were being done in groups, and page in operations were being done one page at a time.

pages for Win32 applications and DLLs from their original executable files. This setup doesn't entirely rule out the possibility of using self-modifying code: if a code page is modified (usually by a debugger), the page becomes part of the process's swappable private memory—so it isn't subsequently reloaded from the .EXE. *WriteProcessMemory()* is the API that debuggers can use to modify an application's memory image. Applications can use this API themselves and achieve the same effect.

To assist in the management of all the different types of memory, every active page—that is, every page that is part of an executing system module or application—has a handle to a *pager descriptor* (*PD*) stored with it. A PD holds the addresses of the routines used to move a page back and forth between physical memory and the disk. Regardless of the type of memory the page contains, to get the page into or out of memory the physical memory manager simply calls the appropriate function as defined by the page's PD. Figure 4-5 shows the structure of a PD. A page is defined as a "virgin" page if it has never been written to during its lifetime. (Win32 application code pages are usually virgin pages, for example.) A page is "tainted" if it has been written to at least once since it was originally allocated, and a tainted page is either "dirty" or "clean" depending on whether it has been written to since it was last swapped into physical memory—in which case its contents must be written out to the swap file before the physical memory page can be re-allocated.

```
typedef ULONG FUNPAGE ( PULONG ppagerdata, // Page data word
                        PVOID ppage,       // Pointer to page
                        ULONG faultpage);  // Page #
typedef FUNPAGE * PFUNPAGE

struct pd_s {
    PFUNPAGE pd_virginin;    // Swap in a virgin page
    PFUNPAGE pd_taintedin;   // Swap in a tainted page
    PFUNPAGE pd_cleanout;    // Swap out a clean page
    PFUNPAGE pd_dirtyout;    // Swap out a dirty page
    PFUNPAGE pd_virginfree;  // Free a virgin page
    PFUNPAGE pd_taintedfree; // Free a tainted page
    PFUNPAGE pd_dirty;       // Mark a page as dirty
    ULONG pd_type;           // Page is swappable or not
};
```

Figure 4-5.
Pager descriptor structure.

For a Win32 code page, only the *pd_virginin routine* is needed—all others are null operations. For a Win32 data page, the PD functions would be set up this way (a null entry denotes, esssentially, a no-op):

pd_virginin	Load the page from the .EXE file.
pd_taintedin	Load the page from the swap file.
pd_cleanout	Null.
pd_dirtyout	Write the page to the swap file.
pd_virginfree	Null.
pd_taintedfree	De-allocate the page's space in the swap file.

The functions for an initialized swappable data page would be the same as this except that the *pd_virginin* routine would point to a routine that zero fills the page.

In Windows 3.1, the system allocates the swap file during system setup. This allocation involves the user in responding to a few rather obscure questions, and once it is created, the swap file occupies a sizable chunk of the hard disk. Regardless of what the system actually ends up using, the swap file stays the same size, and Windows 3.1 doesn't offer the user much help in tuning its size to the minimum necessary amount of memory. Windows 95 fixes these deficiencies by using a normal disk file (not hidden, not contiguous) that expands and contracts to the required size during system operation. The swap file gets only as large as it has to, and the user is never involved in either setting it up or adjusting its size.

The bad news about this technique is that under certain conditions the swap file can become much larger than it has to be. For example, if you run one application, get a lot of its data pages dirty, and then run a second application, the first application's data pages will swap to the front of the swap file. Now, if you dirty up plenty of data pages in the second application, switch back to the first application (forcing those data pages out to the end of the swap file), and quit the first application, there will be an unused hole at the front of the swap file.[18] One feature of the Windows 95 design that helps reduce this fragmentation problem is that a physical memory page doesn't always

18. Although it wasn't implemented in the test releases, Microsoft planned to incorporate a background swap file compaction process to prevent the swap file from growing too large.

occupy the same page in the swap file. Unlike in Windows 3.1, if a dirty page has to be swapped out in Windows 95, it's swapped to the first available page in the swap file. This tends to push pages toward the front of the file.

Virtual Memory Management

Virtual memory management in Windows 95 did get considerably more complex. Windows 95 puts several new demands on the virtual memory manager:

- The new Win32 application type with many new API functions that support a number of different shared and dynamically allocated application memory types
- Dynamically loadable system components

All of these demands require changes to the 32-bit protected mode virtual memory manager, although no changes are required to support the older Win16 applications. First, let's examine the new virtual memory types that Windows 95 must support for Win32 applications. As you can see in Figure 4-6 on the next page, Windows 95 allows a Win32 application to consume an enormous virtual address space—and there are plenty of new features available to Win32 programs to encourage the consumption of all that space, including true shared memory and a number of new dynamic memory allocation capabilities. The base OS allocates all Win32 application private virtual memory regions within the lower 2 GB of the virtual address space. All shared memory objects—for example, shared memory regions created by the application—reside within the 2-GB to 3-GB region. Originally, the design had the Windows subsystem DLLs living within this shared memory region. A later change moved these DLLs above the 3-GB boundary, mapping them into the System VM's address space as necessary. Notice that a Win32 application has a true 4-GB address space. Calls to system DLLs are direct calls with no ring transition and no context switch. The advantage of this approach is its speed—there's no overhead beyond the overhead of the function call itself. The disadvantage is that an application can obtain a pointer into the system address space and start poking around—possibly to no good effect. Under Windows NT, the system address space is truly protected and no application can obtain a pointer into it. In this particular instance, the Windows 95 designers went for performance over security.

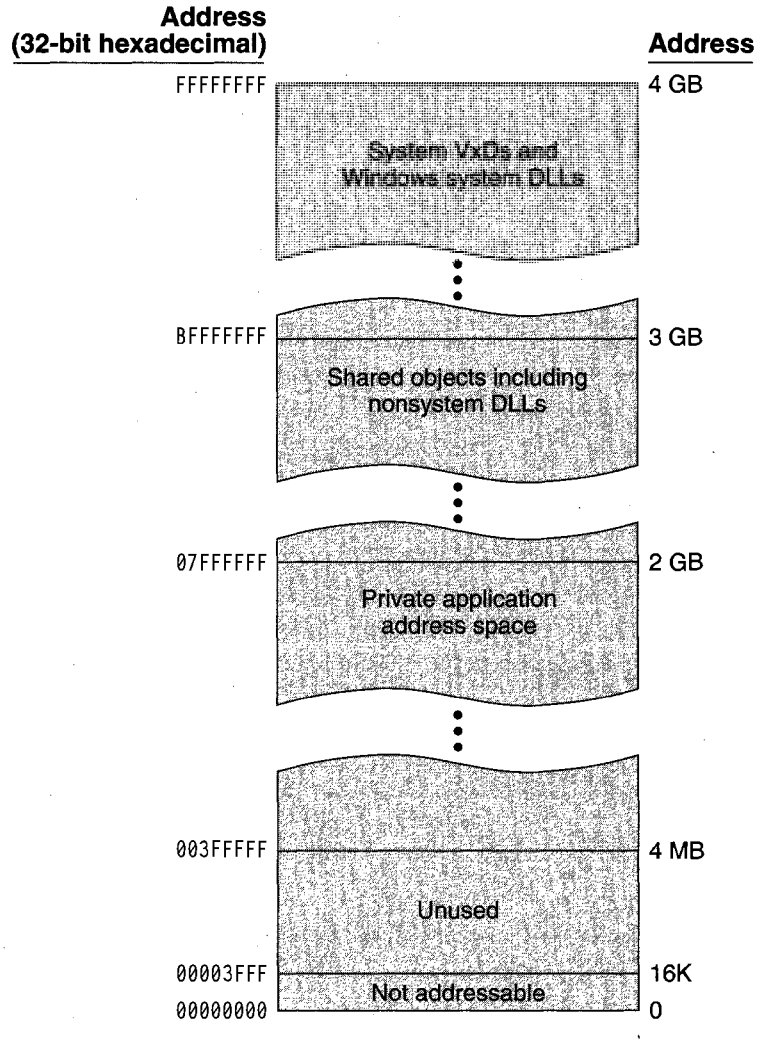


Figure 4-6.
Win32 application virtual memory map.

Within the shared memory region, the different objects will appear in the address space of every Win32 process. This means that whenever the system allocates a shared object, that piece of the address space is reserved in the memory map of every Win32 process—regardless of whether the process cares about the particular shared object. Suppose, for example, that you have two processes A and B that need to communicate with each other by means of a 64K shared memory region.

Process A allocates the shared region, and process B attaches to it. The system determines where in the shared region the memory actually exists. Let's say that the system allocates address 0x8000000 to 0x800ffff. Now suppose that processes C and D run and do some similar allocation and sharing. C and D don't care, or even know, about A and B and their shared memory area. This time the system will allocate a shared region for C and D between 0x8010000 and 0x801ffff. The first 64K has already been reserved across all memory maps, so it's unavailable to C and D. You might object that the disadvantage here is the possibility of filling up the shared region. But seriously? One gigabyte of shared memory? The huge advantage is the performance benefits gained by mapping a shared region to the same address in every process that uses it. A process can access the region by simply using the 32-bit pointer the system hands back—there's not even any system call overhead.[19]

The memory management services within the base OS must support the creation of many different memory object types within the application's virtual address space. Managing their allocation and deallocation efficiently is a key aspect of the system's memory management capabilities. Many virtual memory management functions require that the system back up the virtual memory by allocating physical memory at some point (although there are functions that simply reserve never-to-be-used regions of virtual address space). However, actual physical memory allocation (that is, RAM allocation) may not occur immediately since there's no need to back up the virtual memory until the application touches the memory page. But the system does have to take steps to make sure space is available in the swap file.

Memory Mapped Files

Perhaps the most important new memory management feature for a Windows programmer is the support of shared memory operations through memory mapped files. In fact, this is the recommended way of allocating and using shared memory regions. Typically, applications will use this facility to enable access to large memory resident data structures. To access a memory mapped file, an application must obtain a handle to a *file mapping object* using the *CreateFileMapping()* API function. Once the application has a handle to the file mapping object, it can use the *MapViewOfFile()* API shown in Figure 4-7 on the next page to obtain a memory address for the memory region. Other applications

19. Again, this approach differs from Windows NT's, in which a shared region can appear at different virtual addresses within the memory maps of the processes that use it.

```
LPVOID MapViewOfFile(hMapObject, fdwAccess, dwOffsetHigh,
                     dwOffsetLow, cbMap)

HANDLE hMapObject;    // File mapping object to map
DWORD fdwAccess;      // Access mode
DWORD dwOffsetHigh;   // High-order 32 bits of file offset
DWORD dwOffsetLow;    // Low-order 32 bits of file offset
DWORD cbMap;          // Number of bytes to map

LPVOID MapViewOfFileEx(hMapObject, fdwAccess, dwOffsetHigh,
                       dwOffsetLow, cbMap, lpvBase)
    //
    // First five parameters same as MapViewOfFile
    //
LPVOID lpvBase;       // Suggested starting address
                      //  for mapped view
```

Figure 4-7.
Mapping a file into memory.

can access the same file mapping object using the *OpenFileMapping()* and *MapViewOfFile()* APIs.

The pointer returned by *MapViewOfFile()* is a virtual address somewhere within the 2-GB to 3-GB region. As you'd expect, there's no predicting where the memory object will be within this region, but the shared memory region will appear at the same virtual address within different processes.

The *MapViewOfFileEx()* API, also shown in Figure 4-7, is usable in Windows 95. This API tries to force the system to allocate a shared region at a particular address (and it will fail if some part of that address space is already in use). Under Windows NT, this explicit request is necessary since the system won't guarantee the same virtual address for the shared region in each process. Under Windows 95, *MapViewOfFileEx()* is redundant.

Reserving Virtual Address Space

An application can reserve a region within its virtual address space using the *VirtualAlloc()* API (Figure 4-8). The address the application passes as a parameter may be a specific address, or the application may simply request a region of a certain size at any available address. The application can simply reserve the virtual address space—meaning that no physical memory is ever allocated to back up the virtual memory. The application can also set certain conditions on the region, such as read-only protection.

```
LPVOID VirtualAlloc(lpvAddress, cbSize, fdwAllocationType,
                    fdwProtect)

LPVOID lpvAddress;          // Virtual address of region
DWORD cbSize;               // Size of the memory region
DWORD fdwAllocationType;    // Type of allocation
DWORD fdwProtect;           // Type of access protection
```

Figure 4-8.
Reserving a virtual address region.

Private Heaps

An application can take advantage of the existing memory allocation capabilities of the system by creating private heap space using the *HeapCreate()* API (Figure 4-9). Once the application has a handle to the heap area, it can allocate memory from the private heap in the same way it allocates memory from the Windows global heap. The system reserves the memory for the heap within the private virtual address region of the application and won't allocate physical memory to back up the virtual memory until it's needed.

```
HANDLE HeapCreate(flOptions, dwInitialSize, dwMaximumSize)

DWORD flOptions;        // Heap allocation flag
DWORD dwInitialSize;    // Initial heap size
DWORD dwMaximumSize;    // Maximum heap size
```

Figure 4-9.
Private heap allocation.

Virtual Machine Manager Services

The Virtual Machine Manager is the single most important operating system component in Windows 95. As distributed, the VMM is actually a VxD that lives in the DOS386.EXE file together with a number of other VxDs, such as the Plug and Play subsystem and the filesystem drivers. This combination of VxDs forms the base operating system for Windows 95. Once it is loaded during system initialization, the VMM is permanently resident. Although the VMM uses the binary format of a VxD, it certainly isn't a virtual device driver in the sense in which you normally regard VxDs.

Every VxD can define a *service table* to identify entry points to functions within the VxD that provide a *service* to other VxDs or applications. You can think of the services provided by a VxD as an API that's internal to the operating system. Since you can add VxDs to a system and write applications that call on VxD services, for some purposes you can consider these services as an extension to the Windows API. No, that doesn't mean that the Windows 95 API suddenly grew by several hundred functions. The services are for use by other VxDs when they're running at ring zero. Calling them indiscriminately from an application guarantees a system crash. VxDs don't have to provide any services, though, and there are standard system VxDs that don't. However, Windows 95 does include a documented interface that allows applications to call VxD services, and therein lies the major difference from Windows 3.1, which included only an undocumented and nonportable interface.

The VMM actually provides a central core of services callable by any VxD and doesn't deal specifically with any device. Windows 95 contains over 700 services within the base OS. The fact that the VMM provides close to half of these is an indication of the relative importance of the VMM.[20] Normally, the use of VMM services is the domain of device drivers, debuggers, and other system-level extensions to the base OS, and the scope of VMM services covers the lowest level of OS requirements, such as

- Memory management—meeting the physical and virtual memory allocation requirement details
- Scheduling—dynamic priority management and timeslice administration
- Interrupt handling—hardware device and fault management
- Event coordination—notification and thread supervision

As its name suggests, the VMM controls Windows' virtual machines. It keeps track of each VM using a *VM control block* and a 32-bit handle that identifies the specific VM. (The handle is actually the virtual address of the VM control block.) The VM control block contains information about the current state of the VM, including the VM's execution status (idle or suspended, for example), the VM's scheduling

20. A normal Windows 3.1 system includes a total of about 400 services, and the Windows 3.1 VMM offers 242 services.

priority, and copies of the VM's registers. Discussions of VMM services refer to VMs as *clients* of the VMM, and you'll often see references to "client" data structures such as the *Client_Reg_Struc* area used to save the VM's registers.

Calling Virtual Machine Manager Services

Before looking at how VxDs and applications interact with the VMM, we should look at how the OS supports the various code paths in the system, noting at the same time several new features of Windows 95. Developing a VxD is not a trivial task. The VMM and every other VxD is always 32-bit protected mode software running at ring zero, and you have to use assembly language to call upon VMM services. The Microsoft Windows Device Driver Kit tells you why you might want to do this and how to go about it.

Figure 4-10 shows an example call to the VMM's *Call_Global_Event* service. As you can see, the *VMMcall* macro masks the true nature of the call to the VMM service. A *VxDcall* macro is used in a similar way. In fact, both macros generate the same sequence of instructions, so the difference in name is more for documentation than for any other reason.

```
include    vmm.inc                          ; All the basic
                                            ;  definitions
        ; Start of fragment
mov     esi, OFFSET32 CallbackProc      ; Address of callback
mov     edx, OFFSET32 CallbackData      ; Address of data
VMMcall     Call_Global_Event
        ; End of fragment
```

Figure 4-10.
Calling a VMM service from a VxD.

VMM Callbacks

One of the important techniques used by the VMM and other VxDs is a *callback* mechanism that allows a VxD to register the address of a procedure for the VMM to call when certain conditions hold. The technique is similar to the way an application registers window procedures that the Windows subsystem calls for message processing. The VMM uses callbacks extensively to notify VxDs of system events such as hardware interrupts and general protection faults and for scheduling related events. Usually, every VMM service that allows the registration of a callback is matched by another service allowing the caller to cancel the callback.

The *Call_Global_Event* service illustrated in Figure 4-10 is one of the services that use a callback procedure. When the VxD makes the call to this service, the VMM will make arrangements to call the procedure whose address is supplied to the VMM service (*CallbackProc* in the example) and pass it the other parameter (*CallbackData*) supplied in the original request to *Call_Global_Event*. In this particular example, the callback from the VMM may happen immediately, or if the VMM is busy with a hardware interrupt, it will defer the callback until the interrupt processing is complete. Thus, when the VMM calls the VxD's callback procedure, the VxD knows the current status of the system and has a reference to some data that identifies the purpose of the call.

An important point to note about the VMM in general and the callback mechanism in particular is that many VxDs can call the same service. If the VMM registers more than one callback for a particular service, it simply works its way through a list, making each callback in turn. If you need exclusivity, you have to arrange to get it some other way.

Another example of a callback is the VMM's *Call_When_Idle* service. When the system is completely idle—that is, when there is no Windows action and no VMs are running—the VMM will call every VxD that registers itself with the *Call_When_Idle* service. Idle time is a good time to consume processor cycles for housekeeping chores. Windows 3.1 used it for writing modified memory pages out to the paging file. Windows 95 uses it for swap file compaction. Other VxDs could register a callback for their own idle purposes. But on a busy system there are only small amounts of idle time and no guarantee of when they'll occur or which other VxDs they may have to be shared with. This indeterminance is an aspect of many callback services—so design accordingly.

Loading VxDs

Windows 3.1 loads VxDs at only one time: during system initialization. There's no provision for loading VxDs while the system is running, as there is for loading application DLLs. Even if a VxD provides only infrequently used services, it must be loaded at startup and remain resident while Windows runs. Since Windows 3.1 uses the SYSTEM.INI file to specify the VxDs to load, installing a new VxD requires the addition of an entry to SYSTEM.INI. Another shortcoming of using VxDs in earlier versions of Windows was the identifying mechanism used by the system. Every VxD had to have a unique identifier, and VxD developers had to apply to Microsoft for this magic number.[21] The developer then

21. Internet e-mail to *vxdid@microsoft.com* will get you the information you need. One ID actually gives you the ability to create up to 16 unique VxDs.

embedded the so called VxD ID within the VxD, and the system used this number at runtime to connect VxD service callers to the correct VxD. There's nothing sinister about having to apply for an identifier; it's simply an artifact of a rather primitive method for guaranteeing uniqueness.

Windows 95 solves most of these problems. VxDs are dynamically loadable and unloadable, and for most VxDs a new naming convention does away with the need to acquire a private identifier. Microsoft's shorthand for dynamically loadable VxDs is "dynaload VxDs," or simply "DL VxDs." For brevity, we'll call them "DL VxDs." The operating system loads DL VxDs into the system's private virtual address space (above the 3-GB boundary), and the DL VxD author can identify the regions of the VxD's code and data that are pageable. This identification of pageable regions allows the developer to optimize the DL VxD's working set. Also, in Windows 95, applications can cause the system to load DL VxDs by name, which eliminates the need to edit the system's configuration files. You need a unique VxD ID from Microsoft only if the VxD offers VxD services or other API functions. If your VxD doesn't do this, you can simply use the constant *Undefined_Device_ID* as its identifier. Windows 95 will happily load multiple VxDs with this identifier.

For compatibility, you can still load VxDs during system startup. In fact, that's what happens with the VMM and most of the base system VxDs. If you write a VxD for disk device support, for example, you'll probably want it to be loaded during system boot. If the presence of your VxD is required only occasionally, the dynamic loading technique is the one to use. Network support is a good occasion for the use of DL VxDs, notably for the large components such as network transports. The Windows 95 Plug and Play and installable filesystem components are themselves dependent on DL VxDs. The dynamic loading of VxDs is the domain of the VXDLDR module—itself a (static) VxD. VXDLDR offers six services callable by other VxDs or indirectly by application programs.

The general rules for a VxD in earlier versions of Windows specify both its executable format and a number of interfaces it must support.[22] The system uses the mandatory interfaces to allow VxD initialization and to call the VxD with certain systemwide events the VxD must respond to. There are several events associated with system initialization and shutdown, for example, that each VxD is asked to process.

22. The only substantive change to the executable format of VxD in Windows 95 is that you can now define both memory resident and pageable code and data sections.

The rules for DL VxDs don't change very much in Windows 95. The format is a little different, and there are some restrictions on what a DL VxD can do. Only one restriction is significant: if a DL VxD offers any VxD services, it can't be dynamically unloaded. One reason for this restriction is the difficulty of notifying other VxDs or applications that a service they're using is about to disappear. Consider the problem associated with removing a DL VxD that provides a callback service that other modules might yet try to use.[23]

The Shell VxD

The final piece in the VMM puzzle is the module called the *Shell VxD,* or sometimes the *shell device.* Note first that the Shell VxD has absolutely nothing to do with the user shell, the application that manages the desktop. Once again, overloaded terminology can lead to confusion. The Shell VxD is the last component of the base system that gets loaded, and it's responsible for loading the Windows subsystem (Kernel, User, and GDI). As the user shell is to the user, so the Shell VxD is to the ring three software.

There's a Shell VxD in earlier versions of Windows as well. One of its main functions was the display of dialog boxes on behalf of a VxD. It's the Shell VxD that generated the *System has become unstable* dialog that came up frequently in Windows 3.0 and only occasionally—rarely—in Windows 3.1. Windows 95 expands the Shell VxD services considerably, adding functions in two areas that are relevant to this discussion.

The Shell VxD manages to do its dialog box work by running briefly within the context of an application. Its memory mapping and resources are those of the System VM, and in some senses the Shell VxD masquerades as a Windows application to display a dialog. Windows 95 generalizes this facility and adds Shell VxD services that allow a VxD to run at application time. A VxD entered at application time can do anything an application can: open files, load DLLs, and send messages, for example. VxDs achieve application time execution by scheduling an event using the Shell VxD's *_SHELL_CallAtAppyTime* service.[24] Windows 95 implements application time by providing an application thread that the VxD runs on during the callback. Application time isn't

23. No doubt those who probe around in the depths of Windows will soon come up with ways to overcome this restriction.

24. The service mnemonic gives away the name genealogy. Its originator called this context "appy time"—a play on "application" and "happy." Unfortunately, Windows isn't allowed to be whimsical, so "application time" is what the name became.

always available: during system initialization and shutdown, for example, the system is in a state in which it can't support application time processing. One use of application time is to post a graphical Windows dialog informing a user of the options when he or she has pressed Ctrl+Alt+Del to close a nonresponding application. In Windows 3.1, the system could only display a character mode blue screen.

Right about now you're probably beginning to see the expanded possibilities in Windows 95 for applications to interact with the base OS. It gets better yet. The Windows 95 Shell VxD also offers three new services that deal directly with Windows messages:

_SHELL_PostMessage posts a message to a specified window.

_SHELL_BroadcastSystemMessage sends a message to a specified list of windows and VxDs. This service is the same as the Windows 95 *BroadcastSystemMessage()* API.

_SHELL_HookSystemBroadcast allows a VxD to monitor calls to the *_SHELL_BroadcastSystemMessage* service, so that even if a particular VxD is not a target of the broadcast, it can still observe the message.

The windows and messages involved in these new services are exactly what you'd expect: application window handles (the *hWnd*s in an application) and the message identifier and message parameter (the *wParam* and *lParam* in an application message loop). Because the Shell VxD doesn't constrain the message parameters in any way, you can use the *_SHELL_PostMessage* service to set up private transactions between a VxD and an application. It's essentially a clean way for system components to send messages to applications.

Getting Around in Ring Zero

OK, enough discussion of the superstructure. It's time to see how all these pieces collaborate. Of the more interesting paths in the Windows code, the hyperspace jumps between ring three and ring zero and some of the trails within ring zero are among the most revealing. Figure 4-11 on the next page illustrates the variety of different call and return transitions. All are code paths executed as a result of a function call—either a Windows or an MS-DOS API or a call to a base system service. Other paths, taken as the result of hardware interrupts or page faults, aren't illustrated in Figure 4-11.

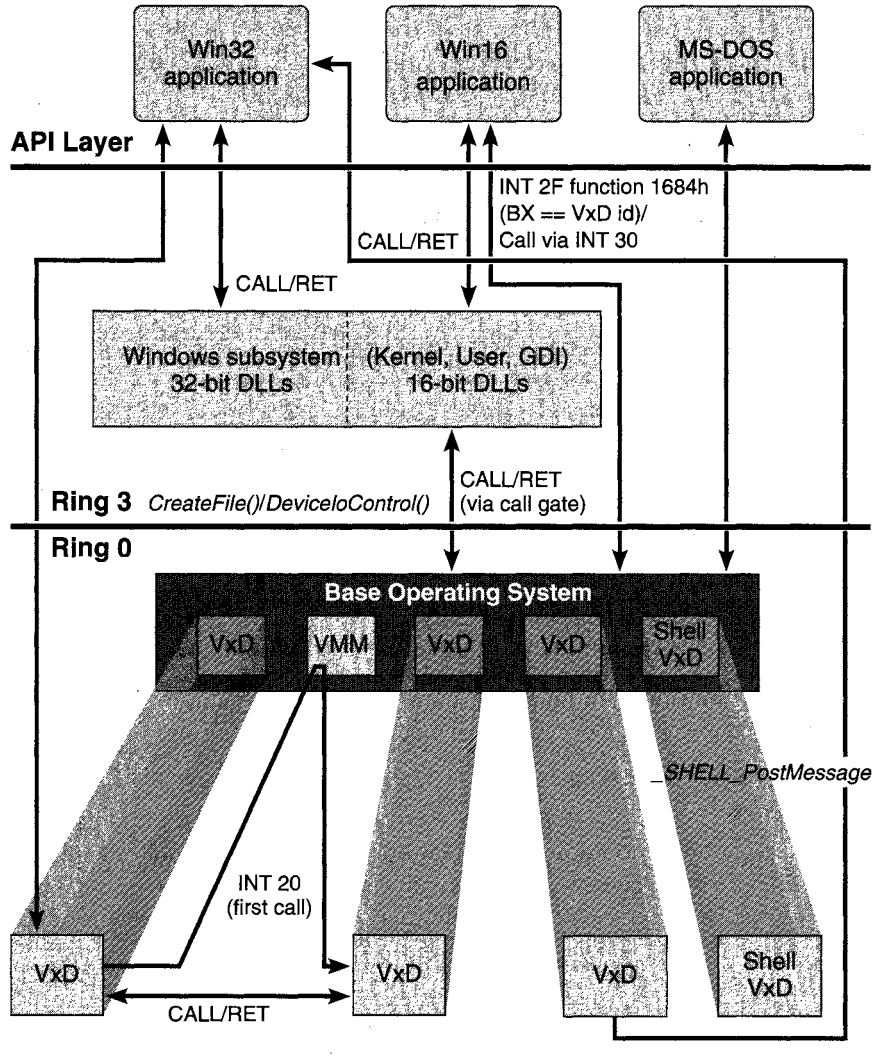


Figure 4-11.
Calls and returns among applications and VxDs in Windows 95.

In Windows 3.1, both MS-DOS applications and the Windows DLLs issued INT 21 software interrupts to call on system services as a result of API calls from applications. Ultimately, these INT instructions caused a general protection fault that the Windows 3.1 VMM picked up in ring zero. In the case of the system virtual machine, the base OS would then switch the VM to virtual 8086 mode and—for all VMs—the MS-DOS operating system code would run to process the API call.

Also illustrated in Figure 4-11 is the INT 2F interface supported by Windows 3.1. For compatibility's sake, the INT 2F interface still works under Windows 95. But that isn't the way you should do it anymore. The Windows 3.1 INT 2F function 1684h interrupt allows an application to retrieve an entry point address for a VxD service. The additional parameters in the call have to specify the VxD identifier. The INT 2F call results in a fault that the VMM intercepts. Using the VxD identifier, the VMM searches for a matching VxD and if successful returns an address that allows the application to directly call the VxD, requesting one of its services. Windows 3.1 actually implements this call by giving the application the address of an INT 30 instruction within a memory segment full of INT 30s. When the application calls the INT 30, there's a fault. The VMM picks up the fault, recognizes it as an INT 30 request, figures out the offset of the particular INT 30 within the segment, and, lo and behold, there's the index to the requested VxD service. Barring some trickiness in returning to the application, this interface works the same for both Windows and MS-DOS applications.

Calling Windows 95 Base OS Services

Obviously, Windows can't do anything about the fact that MS-DOS applications use INT 21 to call system services. File I/O–related calls now get handed directly to the protected mode filesystem INT 21 handler, and the entire filesystem transaction executes in protected mode. The Windows subsystem no longer issues software interrupts to initiate the trap from ring three to ring zero—the subsystem now uses a 386 call gate, passing parameters that identify the required ring zero service. This is a faster operation than trapping and unraveling a GP fault and results in a small performance gain. The return from ring zero to ring three is similarly elegant, simply using a return via the call gate. In the case of the System VM, there is no excursion into virtual 8086 mode—the processor remains in 32-bit protected mode throughout.

Although Windows 95 still supports the INT 2F interface for compatibility's sake, the recommended interface now uses the Win32 API functions *CreateFile()* and *DeviceIoControl()*. If you're familiar with Windows NT, you may already have seen these APIs. *DeviceIoControl()* in particular is intended for use as a general purpose interface that allows private communication between an application and a device driver. Windows 95 uses the interface both for device control and for communication between applications and VxDs.

To initiate communication between an application and a VxD, the application must obtain a handle to the VxD. You use the *CreateFile()* API function to do this (Figure 4-12). The naming syntax for the VxD is a little unusual. To get a handle to the Shell VxD, for example, you use the string "\\.\SHELL" as the filename in the *CreateFile()* call. This naming syntax works for any VxD registered with the system.

```
HANDLE CreateFile(lpszName, fdwAccess, fdwShareMode, lpsa,
                  fdwCreate, fdwAttrsAndFlags, hTemplateFile)

LPCTSTR lpszName;              // Pointer to filename string
DWORD fdwAccess;               // Access mode (read/write)
DWORD fdwShareMode;            // Share mode
LPSECURITY_ATTRIBUTES lpsa;    // Address of security
                               //  descriptor
DWORD fdwCreate;               // Creation mode
DWORD fdwAttrsAndFlags;        // File attributes
HANDLE hTemplateFile;          // Handle to file with
                               //  attributes to copy
```

Figure 4-12.
The CreateFile() *API function.*

Figure 4-13 shows the API definition for the *DeviceIoControl()* function. In its normal mode, the API uses the device control code to initiate a device-specific operation—formatting a floppy disk, for example. When the function is used for communication with VxDs, the device control code and the contents of the input data buffer and the output buffer are entirely application defined. To fully support a VxD interface for general application use, the VxD developer will have to publish the supported control codes and the other details of the data exchange protocol. But if you write both the application and the VxD, you can use *DeviceIoControl()* as a private interface for communication between ring three and ring zero software.[25] Within the system, the VMM System Control service, which is called with a W32_DEVICEIOCONTROL message, dispatches the *DeviceIoControl()* call to the target VxD.

Calling from One VxD to Another

The last interaction we'll look at is the call and return mechanism between VxDs that's used within the base operating system. The method

25. The *DeviceIoControl()* interface also has the advantage that, for published functions, it's portable between Windows and Windows NT. An INT 2F interface definitely isn't.

```
BOOL DeviceIoControl(hDevice, dwIoControlCode, lpvInBuffer,
                     cbInBuffer, lpvOutBuffer, cbOutBuffer,
                     lpcbBytesReturned, lpoOverlapped)

HANDLE hDevice;              // Handle of the device
DWORD dwIoControlCode;       // Operation control code
LPVOID lpvInBuffer;          // Input buffer address
DWORD cbInBuffer;            // Input buffer size
LPVOID lpvOutBuffer;         // Output buffer address
DWORD cbOutBuffer;           // Output buffer size
LPDWORD lpcbBytesReturned;   // Address of actual output data
LPOVERLAPPED lpoOverlapped;  // Address of overlapped
                             // structure
```

Figure 4-13.
The DeviceIoControl() *API function.*

relies on the system's ability to create a unique 32-bit number formed from the VxD identifier and the VxD's service number. In Windows 3.1, the VxD identifier had to be assigned before link time. In Windows 95, the dynamic VxD loading mechanism allows the VxD identifier to be determined at runtime. Both the *VMMcall* and *VxDcall* macros generate code that contains an INT 20H instruction followed by the 32-bit number identifying the required VxD and service. At runtime, the INT instruction causes a fault that's picked up by the VMM. The VMM examines the VxD service identifier embedded in the code and replaces it with a direct CALL to the VxD service entry point. Subsequent calls to the VxD then go directly rather than cause a fault.

Dynaload VxDs use a similar mechanism in Windows 95, but there are some subtle differences:

- At compile time, the *VMMcall* macro generates a CALL instruction. The target of the call is an external symbol in the target VxD indexed by the service identifier.
- At load time, VXDLDR replaces this call with an INT 20 instruction followed by a 32-bit word containing the module identifier and VxD service number. VXDLDR also sets the high bit in the 32-bit word to denote that this is a call from a dynaload VxD.
- At runtime, the VMM patches the INT instruction, using the 32-bit word in the code to map the module identifier to a VxD identifier.

You can see another reason for the no-services restriction on dynaload VxDs. Since the VMM patches the calls to VxD services to actual CALL instructions, if a target VxD were unloaded the VMM would have to go around changing all the CALLs back to INTs.

VMM Service Groups

The VMM is by far the dominant provider of base operating system services, and many of the services are either new or improved for Windows 95. Base OS support for the new threaded architecture for Win32 applications called for many changes and additions to the services, including thread management, scheduling, and mutual exclusion primitives. The largest single category of VMM services (about 20 percent of the services) deals with memory management. Other services are split among several different categories. In this section, we'll look briefly at the various service groups. All of these services are offered by the VMM.

Event services allow the caller to register callback procedures for global events or events for specific virtual machines. Windows 95 adds support for thread events—allowing a VxD to signal an event for a specific thread.

Memory management services include many different memory allocation and de-allocation functions for both physical and virtual memory. Other services that provide information about memory conditions support the memory management functions. Windows 95 adds services that support the creation and management of memory for Win32 applications.

Nested execution and protected mode execution services provide the ability for a VxD to call software within a specific virtual machine that's running in either virtual 8086 mode or protected mode. The system may need to call an MS-DOS real mode device driver or TSR, for example—both of which are always executed in virtual 8086 mode.

Registry services are new for Windows 95. They allow VxDs to interrogate the contents of the on-disk registry. The VMM registry services are similar to those available to applications via Windows API functions.

Scheduler services let a VxD influence the operation of both the primary scheduler and the timeslice scheduler. The VxD's influence can include creation and destruction of individual threads and VMs and adjustments of the current scheduling priorities and timeslice parameters.

Synchronization services offer a range of functions for managing semaphores and *mutual exclusion objects* (*mutexes*). The VMM also offers a number of associated services related to critical section management. Mutex object management, thread-specific services, and several of the critical-section services are all new in Windows 95.

Debug services have been improved in Windows 95, toward the goal of providing better base OS support for system-level debugging tools.

I/O trapping services provide a way for VxDs to collaborate with the VMM to manage the processor's I/O ports. Using these VMM services, a VxD can control access to individual I/O ports.

Processor fault and interrupt services allow VxDs to involve themselves in the system's handling of specific global conditions such as page not present faults and NMI interrupts.

VM interrupt and callback services interface a VxD to the software and hardware interrupt status of an individual VM. For example, a VxD can acquire and modify current interrupt vector settings within a specified VM.

Configuration manager services interface a VxD to the Plug and Play subsystem incorporated in Windows 95.

Miscellaneous services cover a host of other functions used to support VxD execution, including queries about system initialization, error handling, linked list manipulation, time-outs, and even internal versions of the faithful *printf()* function.

Application Support

Although the details of an operating system can be a fascinating study in and of themselves, the OS must ultimately be judged on how well it runs application programs and the associated subsystems. In Chapter

Six, we'll look at the details of the subsystem that supports the graphical environment for Windows applications. Here we'll examine the underpinnings for this support. Earlier we looked at the various code paths between the ring zero and ring three components of Windows 95 and at how an application calls directly on the services of a VxD. Windows 95 introduces support for 32-bit Windows applications using Microsoft's Win32 API while it continues support for existing 16-bit applications (nowadays referred to as "Win16" applications).

Unlike Windows NT, which began life as a 32-bit operating system, Windows has evolved slowly toward full 32-bitness. Ever since the release of Windows/386 in 1988, Windows has included 32-bit code that exploited the 386. Initially, this code was confined to the ring zero system components. Then, in the era of DOS extenders, we saw the first 32-bit applications. Third party VxDs followed. The Win32 API is the next step toward full 32-bit operation for Windows. Win32 is Microsoft's strategic system interface. Its first appearances were with Windows NT and in the subset Win32s API introduced for Windows 3.1. In Windows 95, we see the implementation of this 32-bit API for a product that will most likely sell millions of copies—so, yes, it's pretty important to learn about it. But Windows 95 doesn't support a 32-bit API exclusively. Microsoft hopes that every new Windows application will be a 32-bit application. However, given the sheer number of Windows applications now available, even the most optimistic marketeer has to acknowledge that 16-bit application support is going to be a feature of Windows for some time to come.

The API Layer

The code path from a Windows 95 application to the supporting system code and back is very similar to the one traveled by an application running on Windows 3.1. The system makes extensive use of dynamic link libraries to provide the necessary code paths between the application and the Windows subsystem. Earlier the interface between Windows applications and the Windows subsystem was characterized as a simple call and return interface (Figure 4-7). It might be simple if every system module and application were 32-bit code, but it's actually a lot more complex.

If you think about the Intel processor architecture for a moment, you'll realize that the internal code structure of 32-bit Windows applications and the system code to support them has to be fundamentally

different from the existing 16-bit environment. In particular, the variation in addressing modes means that you can't easily mix 16-bit and 32-bit code. For Windows 95 applications, this means new compilers, assemblers, and linkers to enable 32-bit development. The system itself must at least provide co-resident 32-bit versions of the Windows subsystems (Kernel, User, and GDI) to support the new 32-bit API—alongside the 16-bit API for the older applications. And of course all the code must be small, fast, well tested, and well documented. No problem? Let's see about that.

Mixing 16-bit and 32-bit Code

The problem of mixing 16-bit and 32-bit code has occupied many developers at Microsoft. They have tried various implementation techniques in various forms in earlier versions of Windows and OS/2 and in Windows NT. The Windows 95 implementation certainly represents the state of the art. Whether it's the final word on the subject is a different matter. Here are the problems: [26]

- 32-bit code deals in 32-bit linear addresses (usually called *0:32 addressing*). 16-bit code uses a 16-bit segment selector and a 16-bit offset (known as *16:16 addressing*). There has to be a translation between the two address formats so that the 16-bit code receives valid pointers originally passed as 0:32 parameters—for example, an address parameter that points to a C structure. The solution to this problem involves a technique called *tiling*, in which the system allocates a new 16-bit segment descriptor to describe memory that overlies the memory containing the parameter. (Think of tiles on your roof, and you'll get the idea.)
- In C, the language of choice for Windows, an *int* data type is 32 bits wide in a Win32 application and only 16 bits wide in a Win16 application. When a 32-bit function calls 16-bit code, the 32-bit *int* parameters must be narrowed to 16 bits and then widened on return; this is a relatively easy operation if the parameters are in registers, but many Windows function calls will also push parameters onto the stack.

26. Omitted from this list are some tricky problems associated with the different executable file formats that Windows 95 supports. Essentially, these problems involve the different relocation information contained within the files. There are people who live and breathe object file format issues. We're not going to join them in this chapter.

- 16-bit code will return a 32-bit value (for example, a pointer) in the DX:AX register pair. 32-bit code expects this value to be in the EAX register.
- 32-bit code uses the 386 SS:ESP register pair for stack addressing. 16-bit code uses the SS:SP registers. There has to be a stack switch back and forth and possibly some parameter copying.

An implementation device called a *thunk* is central to the ability to mix 16-bit and 32-bit code effectively.[27] Every call and return from 32-bit code to 16-bit code, or the reverse, requires a thunk. Whenever an API call has to use a thunk, the execution time for the thunk code is pure overhead. If the thunks are slow, application performance suffers. The implementation challenge is therefore to make the thunks consume the smallest amount of memory (remember, there are hundreds of APIs) and the shortest possible execution time. Thunks are always written in assembly language. Figure 4-14 illustrates the different API execution paths in Windows 95 and shows the position of the *thunk layer* relative to the better-known subsystems.

The system handles the stack management issues by building a new stack frame during the transition between the different code types. A call from one code segment type to another will translate parameter formats as the parameter values are pushed on top of the existing stack frame. The addressing of this new frame will then be set up to conform to the rules of the target code type.

Some of Microsoft's previous efforts at thunk design have been documented as parts of various product releases. Windows NT uses a "generic thunk" method whose details you can find in the Win32 Software Development Kit. The Win32s subsystem for Windows 3.1 uses a "universal thunk" mechanism that is an integral part of the subsystem. The Windows 95 thunk method is another iteration and incorporates further execution speed improvements.[28] Some of the speed improvements result from using as much 32-bit code as possible within the

27. The term *thunk* came to Microsoft with one of the original designers of Windows 1.0, courtesy of his college research. It's been around ever since and is now in use as noun, verb, adjective, and insult. Those who were there way back when remember its original definition as "a piece of code that gets you from one place to another."

28. During the development project, the Windows 95 method was sometimes referred to as the "extensible thunk" mechanism, although it may end up with a different final name.

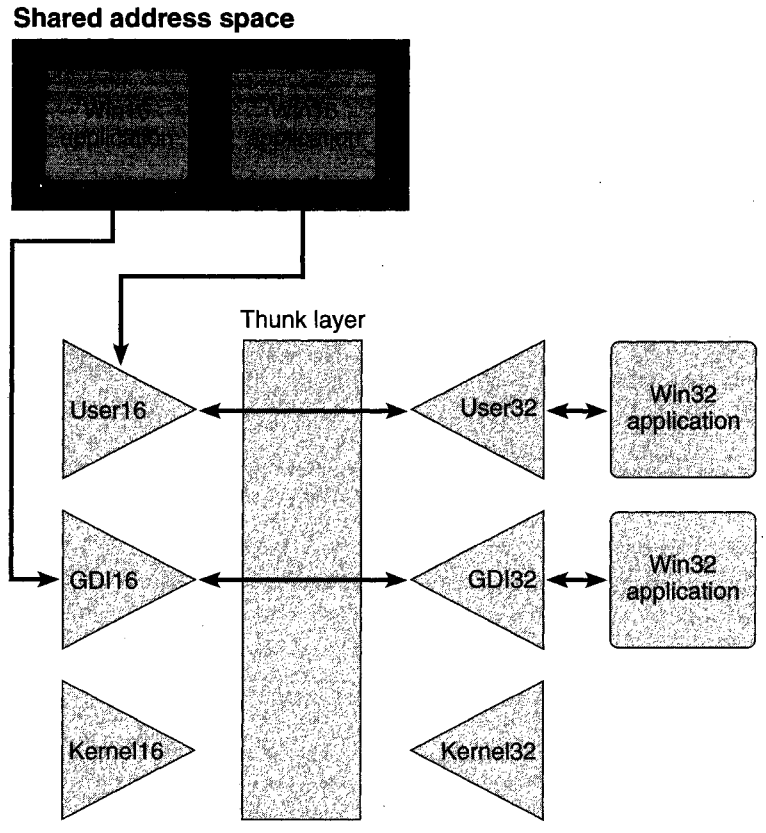


Figure 4-14.
32-bit API support using thunks.

thunk layer. (Microsoft calls this the "flat thunk" mode.) Other speed improvements come from very careful coding of the thunk handler—in particular, minimizing the number of selector loads. (Remember that selector load operations are expensive on the Intel processors since the hardware must validate the new selector against the program's current privileges and memory map.) Late in 1993, the Windows 95 team had a thunk transition down to just seven selector loads and they were still thunking—er, thinking.

Generating large numbers of thunks by hand is a waste of effort, so Microsoft developed some tools to help automate the process.[29] This

29. The thunk compiler toolset became part of the Windows 95 SDK in early 1994.

automation requires the programmer to prepare a description of the source and target APIs in a language that's very close to C, and the result is a sequence of assembly language instructions that form the thunk for the particular API. Figures 4-15 and 4-16 illustrate the input and output for the thunk compiler using the GDI *LineTo()* API function as an example.

```
//
//  LineTo() thunk compiler input. LineTo is:
//
//      BOOL LineTo(HDC hdc, int X, int Y);
//
typedef int INT;
typedef unsigned int UINT;
typedef UINT HANDLE;
typedef HANDLE HDC;

    // 16-bit target followed by 32-bit target
BOOL LineTo(HDC, INT, INT) = BOOL LineTo(HDC, INT, INT) {}
```

Figure 4-15.
Example thunk description input.

```
; 16-bit side
externdef   LineTo:far16

FT_Exit0ThkTargetTable:

    dw    offset    LineTo          ; Entry #55 (say)
    dw    seg       LineTo
    ...

; 32-bit side
public LineTo@12
LineTo@12:
    push    ebp
    mov     ebp, esp
    sub     esp, 40                 ; Workspace for Kernel32
    push    word ptr [ebp+8]        ; Push 3 DWORD to WORD
                                    ;  parameters
```

Figure 4-16. *(continued)*
Example thunk output.

Figure 4-16. *continued*

```
        push    word ptr [ebp+12]
        push    word ptr [ebp+16]
        mov     cl, 50                  ; Thunk index
                                        ; Call 16-bit code and
                                        ;  widen result
        call    QT_Call16_ShortToLong
        leave
        ret     12
```

Note that although Microsoft made the thunk compiler tools available, this technique for mixing 16-bit and 32-bit code is not recommended as a long-term solution. For one thing, the code isn't compatible with Windows NT. If you do choose to use thunks as an interim solution, make sure that the associated code is isolated and easy to replace.

The Win32 Subsystem

You can find the code for Win32 application support in four files in the \WINDOWS\SYSTEM directory:

GDI32.DLL contains the API entry points and support code for the 32-bit graphics engine functions.

USER32.DLL contains the API entry points and support code for the 32-bit window management functions.

KERNEL32.DLL contains the API entry points and support code for the 32-bit Windows Kernel functions.

VWIN32.386 contains a VxD that's responsible for loading the other 32-bit DLLs.

Within these modules lies the complete Win32 subsystem. To get it running, the 16-bit Windows Kernel module will load the VWIN32 VxD the first time there's a call to any 32-bit API. VWIN32 loads the three DLLs and returns to the 16-bit Kernel, which then calls the KERNEL32 DLL initialization function. Once this call is complete, the Win32 subsystem is ready for use.[30]

30. Given that the Windows 95 shell is a 32-bit application, the loading and initialization of the Win32 subsystem will actually occur during the system startup phase.

Most of the code in the 32-bit User DLL is little more than a layer that accepts 32-bit API calls and hands them to its 16-bit counterpart for processing.[31] Although that sounds simple, it's where all the thunk trickery comes in. It's also a sensible way of using tried and trusted code—after all, the 16-bit API implementations have to be there for compatibility reasons. The 32-bit GDI DLL contains a lot of new code and embodies some significant performance improvements. Consequently, the 32-bit GDI handles a lot of API calls directly. The Kernel32 module is completely independent of its 16-bit ally. There is some communication from the 16-bit side to the 32-bit side, but the 32-bit Kernel never calls across to the 16-bit side. This is as you'd expect since most of the code—memory allocation and thread management, for example—is quite different.

Since the call and return between the 32-bit and 16-bit code is a relatively expensive function call, the designers had to look carefully at each API before committing it to the thunk technique. The design guideline was that if the time to execute a 32-bit to 16-bit call and return was a significant proportion of the total execution time for the API call, the API should be replicated in 32-bit code. Examples of these replicated functions are the *Get* functions in GDI such as *GetBrush()* and *GetStockObject()*. These functions simply collect some data and return it to the calling application. Very few instructions are necessary within the API routine. Of course, code replication is out of the question if the API might need to modify a global data structure since the system has to guard against reentrancy problems.

The development team's emphasis on putting their efforts into the new 32-bit code meant that 16-bit applications could pick up many of the benefits. But there had to be a way to get from the 16-bit side to the 32-bit side, so the thunk mechanism also supports calls in this direction. The 32-bit GDI code is in some cases so much better than the 16-bit code that the 16-bit application still runs faster despite the thunk overhead. An example of this benefit is the more efficient 32-bit TrueType rasterizer. Also, to ensure memory allocation consistency, the 16-bit User code calls its 32-bit counterpart to allocate heap storage for 16-bit applications. All the dynamic memory allocation is thus efficiently satisfied from a single 32-bit region.[32]

31. There are actually about 25 User APIs that also exist as 32-bit code. Again, this is for performance reasons.

32. This memory allocation technique supersedes the use that Windows 3.1 made of DPMI in order to get 32-bit memory chunks.

The team was also conscious of the debate that would arise when observers began to analyze the mixture of 32-bit and 16-bit code, so high performance was a priority. The lowest thunk overhead for a Win32 application runs to just over 60 clock cycles, with the average overhead at about 90 clock cycles. For a very expensive API function such as *CreateWindow()*, which has 11 parameters, the overhead is about 100 clock cycles. Windows NT, with its security requirements that call for a ring transition and careful validation of all parameters, imposes a much larger overhead even in a pure 32-bit system call.

Internal Synchronization

One of the biggest design debates inside the Windows 95 development team was over how to deal with system reentrancy.[33] The 16-bit Windows subsystem wasn't originally designed to deal with the possibility of process preemption. Consequently, there are many places in the 16-bit GDI, User, and Kernel modules where the system will fail if one thread is allowed to execute reentrant code concurrently with another. Every operating system has to deal with this problem. Windows NT handles it by blocking threads that try to access the same object at critical times. UNIX and OS/2 contain sections of code that block every thread but one for the duration of a critical section. Windows 95 absolutely required support for the preemptive multitasking of Win32 applications, and since many 32-bit APIs call 16-bit code, the development team had to address the preemption issue. To solve the problem, the team looked at a number of possibilities:

- Develop a new subsystem to support the existing 16-bit applications.
- Use the new Windows subsystem (particularly the GDI module) that the Windows NT team had developed.
- Adopt an approach similar to that of OS/2 2.0, in which each 16-bit Windows application runs as a separate VM—somewhat as the MS-DOS VM support works.
- Use one or more system semaphores to ensure that no more than one thread at a time can run within the 16-bit subsystem.

33. Not only within the development team. During late 1993, this topic became by far the most popular topic of debate in the Windows 95 CompuServe forums and at the various developer events organized by Microsoft.

- Revise the old code to enforce mutual exclusion on system resources within the appropriate critical sections of the 16-bit subsystem (a design technique referred to as "serializing the kernel").

As you can probably imagine, the debate over reentrancy swirled around issues of compatibility, performance, timescale, implementation effort, and long-term value. The different approaches to the reentrancy problem broke down to a question of new code, new architecture, or protection of old code. Let's look at just a few of the specific trade-offs the development team had to take into account as it considered adopting one of the new approaches:

- The nonpreemptive nature of Windows 3.1 and its predecessors has meant that some Windows applications could depend on the ordering and timing of certain system messages. Preempting one of these applications at the wrong time would cause such a program to fail. Breaking this compatibility constraint was simply not an option.
- Application-registered callbacks are another difficult compatibility issue. If the team used a semaphore approach, the procedure for correctly setting the appropriate flags during a callback to a 16-bit application would be a tough one to develop and test; this is a soluble problem, but the solution would have involved huge amounts of testing.
- Rewriting the entire Kernel, User, and GDI subsystems as 32-bit code would have dramatically increased the memory required for the system's working set. The User and GDI modules alone require a working set of about 800K.[34] Measurements indicated that a conversion to 32-bit code would have increased the memory requirement by close to 40 percent, which would have raised the working set requirements for User and GDI to well over a megabyte. Given the goal of running Windows 95 well on a 4-MB system, this increase in memory consumption wasn't acceptable.

34. Out of a planned total working set of around 3 MB for the product—similar to that of Windows 3.1.

- Using the Windows NT subsystem looked attractive but would have required extensive adaptation work for the Windows 95 architecture and a lot more memory to run in. (The Windows NT code is written predominantly in C++, whereas Windows 95 is written in C and assembler.)
- A similar problem would have arisen from adopting the multiple VM solution used by OS/2—more memory would have been needed on the host system. And the OS/2 solution fails to address some critical compatibility issues that the Windows team weren't prepared to ignore.

With radically new approaches disqualified, it came down to figuring out how to introduce protection (by way of mutual exclusion) into the Win16 subsystem. The new 32-bit code designed for the Win32 subsystem simply didn't have this problem: from the outset it was designed to support a multithreaded environment. Each of the potential solutions for the protection of old code traded implementation time off against overall impact:

- A single semaphore guarding the Win16 subsystem against reentrancy would have been the simplest solution. It would have been quick to implement and easy to test, and it would have had no associated compatibility problems. However, under certain conditions it could have had a big effect on the system's multitasking performance.
- Multiple semaphores guarding related groups of Win16 functions would have reduced the adverse effects of a single semaphore on multitasking performance; but when the benefits were weighed against the implementation and testing effort it would require, this design didn't seem to be a compelling solution. Using multiple semaphores to reduce the granularity of a critical section would have imposed a performance overhead. In one measurement, the execution time for a single API increased by 10 percent. Providing the user with a new system that was slower than Windows 3.1 was, again, an unacceptable trade-off.
- The team also looked at a solution somewhere between the single semaphore approach and the multiple semaphores

approach. In this solution, two semaphores would have been used: one for Win16 applications and the other for the 16-bit User and GDI modules. This arrangement would have allowed calls from 32-bit code into the 16-bit User and GDI whenever a Win16 application was doing something else. Unfortunately, this solution would have involved modifying over 1000 entry points within Windows, as well as required modifications to system DLLs and many third party device drivers. Compatibility constraints disqualified this solution too.

- Serializing the Win16 subsystem would have been the most effective solution. Shared resources would have been locked only briefly—minimizing the impact on the system's multitasking performance. Unfortunately, the estimates for implementing this solution indicated that it would have taken a significant amount of time to complete the development work and would have added a massive testing burden to the project. The team realized that the serializing approach would have involved them in one of those software tasks that's virtually impossible to accurately estimate the timescale for until a lot of work has already been completed. Certainly, they knew, months of elapsed time would be involved—enough time to push the product release beyond acceptable limits.

Microsoft decided to adopt the single semaphore solution for Windows 95. Figure 4-17 shows a revised version of the diagram in Figure 4-13, one that depicts what really goes on when 16-bit and 32-bit applications run concurrently. The semaphore that guards the Win16 subsystem against reentrancy is called *Win16Mutex*[35]. This semaphore is set whenever the scheduler hands the processor to any 16-bit Windows thread. The setting of the semaphore has several implications:

- Win32 application threads set and clear the semaphore as they pass through the thunk layer. A concurrent Win32 thread blocks on this semaphore while another thread is executing Win16 code.

35. The awesome power of marketing. *Win16Mutex* used to be *Win16Lock*. After the early technical debates about Windows 95 multitasking effectiveness, the marketing group decided that *Win16Mutex* had fewer negative connotations than *Win16Lock*, and the name was changed.

- A Win32 thread that does not thunk to the Win16 subsystem never blocks on *Win16Mutex*.
- Whenever the scheduler hands control to a Win16 thread, it sets the semaphore. *Win16Mutex* remains set until the Win16 thread yields control.
- The behavior of a 16-bit Windows application will be exactly the same as under Windows 3.1: no preemption and no changes in message ordering, timing, or any other system-dependent operation.

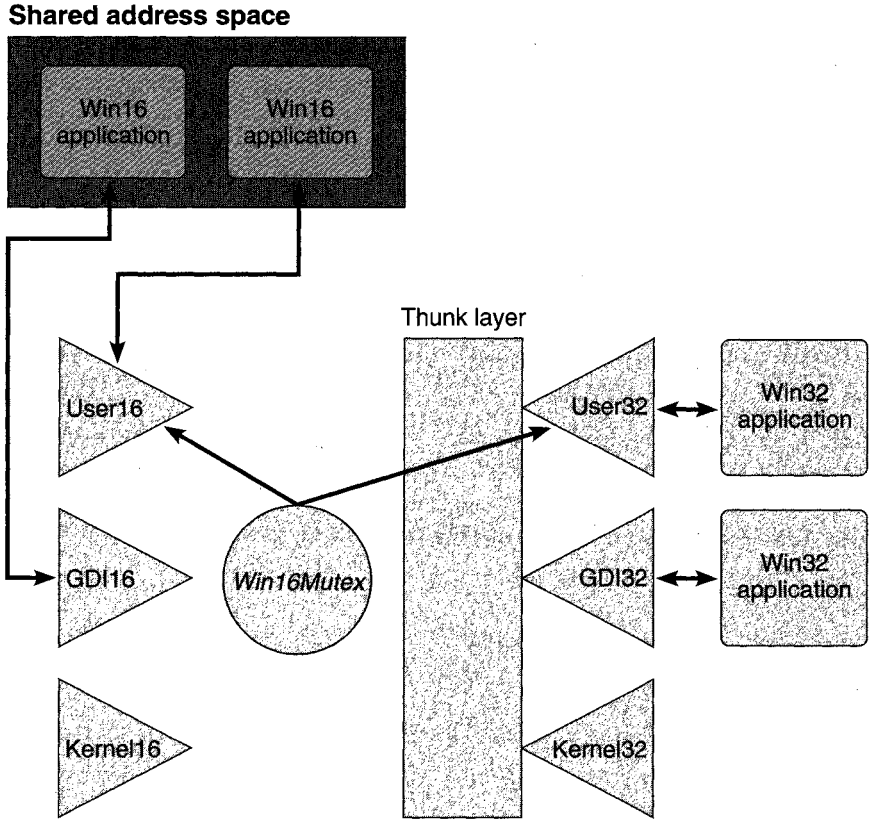


Figure 4-17.
Serializing execution of the Win16 subsystem.

The *Win16Mutex* operations warrant more explanation since they are also the drawback to this solution. Setting *Win16Mutex* prevents a Win32 thread from entering the Win16 subsystem whenever a Win16 thread is active. *Win16Mutex* has to be set because there are nonreentrant Win16 components, such as the common dialog library, that a Win16 application calls directly rather than via an entry to the Win16 subsystem. Setting and clearing *Win16Mutex* as a Win16 thread enters the system won't account for this case, so the semaphore has to remain set whenever a Win16 thread is active. Under normal operation with well-behaved 16-bit applications (that is, with applications that regularly yield control as they should), the effects on the system's multitasking are minimal. At worst, there might be a brief delay in a window repaint for a Win32 application. (And "brief" here is on the order of microseconds.) If a 16-bit application actually hangs up, the system will gradually come to a halt as the Win32 threads block on *Win16Mutex*. When the user hits Ctrl+Alt+Del to get rid of the offending application, the system will reset *Win16Mutex* as part of its cleanup procedure—and everything will proceed normally. If a 16-bit application actually crashes—with a GP fault, for example—then again *Win16Mutex* will be cleared during cleanup. The *Win16Mutex* semaphore is a less than perfect solution—no question. And no doubt critics searching for flaws will pounce on this shortcoming. It is the best solution Microsoft could come up with to the most obvious problem brought about by the compatibility constraints placed on Windows 95. Having examined the trade-offs inherent in each possible solution, I'll happily argue that the Windows 95 designers made the right choice. Ignoring compatibility constraints would have been the worst decision the design team could have made, and the additional constraints of performance, memory occupancy, and project timescale make the single semaphore solution the best one. Windows 95 offers a scheduling mechanism that's markedly better than the one in Windows 3.1 today. Your existing 16-bit applications will run as well as or better than they ever have, you'll get full preemption with new Win32 applications, and in everyday use the combination of the two really won't have a detrimental impact on performance:

- The 32-bit and 16-bit kernel components are independent, so a Win32 thread requesting a potentially lengthy operation, such as file I/O, won't have to call into the 16-bit code.

- The User and GDI calls that do have to grab the *Win16Mutex* semaphore are predominantly ones that have very short execution times, so Win32 threads will need to own the semaphore only briefly. This means that separate Win32 threads will rarely compete for the semaphore.
- Both the shell and the print spooler are 32-bit applications, so the most commonly used components will avoid the problem altogether.

The possible drawbacks to this solution when the user runs a mix of 16-bit and 32-bit applications are another incentive for application developers to concentrate their efforts on Win32 applications. And don't forget: if you truly, absolutely, require a system that provides guaranteed preemption of both 32-bit and 16-bit applications, Windows NT is the product for you.

Conclusion

For students of operating system design, Windows 95 is interesting for its practical implementation of some modern techniques, such as threads. And the base system now fully exploits the 386 processor architecture, with its core components retaining no dependence on 16-bit code or 16-bit processor modes. The ugly practicality of running applications designed for the world's most popular piece of software has meant that some design compromises had to be made. For the purist, the compromises may detract from the major improvements implemented within the base operating system introduced with this version of Windows. For the user and for the developer who's in the business of selling software, the compromises mean compatibility—to this day the only feature guaranteed to increase the popularity of an operating system.

Windows 95 provides the application programmer with some major new opportunities, including the prospect of developing with a full 32-bit API and memory model and the ability to exploit preemptive scheduling. The user will benefit from 32-bit applications in terms of performance, robustness, and increased functionality. Those enhancements won't be the user's first impression, however. That will be provided by the major changes in the appearance of Windows 95, and they're our next topic.

References

Microsoft Corporation. Win32 Software Development Kit. Redmond, Wash.: Microsoft, 1993.

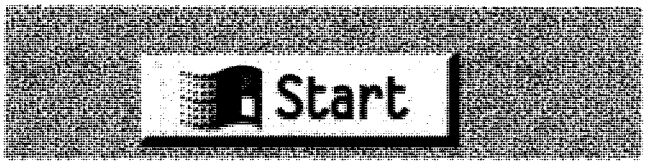
Microsoft Corporation. *Win32s Programmer's Reference Manual.* Redmond, Wash.: Microsoft, 1992.

Microsoft Corporation. Windows 95 Device Driver Kit. Redmond, Wash.: Microsoft, 1994. If you really want to grope around inside Windows, you must have this product. Simply studying the header files reveals a lot of information about the internals of Windows. There are also reams of sample VxD source code if you want to get very serious.

Nu-Mega Technologies, Inc. *Soft-Ice/W Reference Manual.* Nashua, N.H.: Nu-Mega, 1993. If you program seriously for Windows and you don't use this debugger, stop everything and go buy it. Clearly, Microsoft itself was impressed—a preliminary version of Soft-Ice/W for Windows 95 came with the very first external test release. Apart from being a great tool, Soft-Ice/W offers a lot of interesting information about Windows in its product manual.

Oney, Walter. "Mix 16-Bit and 32-Bit Code in Your Applications with the Win32s Universal Thunk." *Microsoft Systems Journal,* November 1993: 39–59. A useful discussion of some of the issues surrounding mixed memory models and thunking techniques.

Pietrek, Matt. *Windows Internals.* Reading, Mass.: Addison Wesley, 1993. If you really want to know how Windows 3.1 does its work, this is a book you have to read. I imagine Matt as he wrote this book as a lone spelunker, flaming torch held high as he crawled through the world's latest and largest heretofore undiscovered system of caves. I hope that Matt is already at work on the version for Windows 95.



CHAPTER FIVE

THE USER INTERFACE AND THE SHELL

Microsoft's introduction of Windows 3.0 in New York on May 22, 1990, was the cornerstone upon which the Windows product line has built an ever increasing market share over the last few years. Although there were many notable new features in the Windows 3.0 release, the product introduction and a large proportion of the product's reviews focused on the improved visual appeal of the Windows interface. Many small, simple improvements to the interface, such as buttons that appeared to move when the user clicked them with the mouse, enhanced the product's immediate appeal—perhaps out of all proportion to their actual importance. The product's eventual success was a function of the other major new features of Windows 3.0 plus Microsoft's intense marketing campaign and the availability of some important new Windows application products. But in the first flush of the product's success, its visual appeal counted for a great deal.

Windows 95 looks as dramatically different from Windows 3.0 (and 3.1) as Windows 3.0 did from its predecessors. From the moment you start Windows 95, you can see that the appearance of Windows has been completely altered. Figures 5-1 and 5-2 on the next page illustrate the difference. Each shows one of the first screens a user sees after initial installation.

So why change a winning formula so completely? Aren't there some major business risks associated with asking a loyal base of users to accept change one more time? Of course there are some risks, and the reception of Windows 95 will determine whether Microsoft's gamble pays off.[1] In this chapter, we'll look at all the new elements of the Windows interface

1. Late in the project Microsoft decided to retain versions of the Windows 3.1 Program Manager and File Manager as desktop accessories for Windows 95—no doubt to lessen the initial shock for experienced Windows 3.1 users.

and in particular at the new Windows 95 shell—itself significantly different from the Windows 3.1 Program Manager.

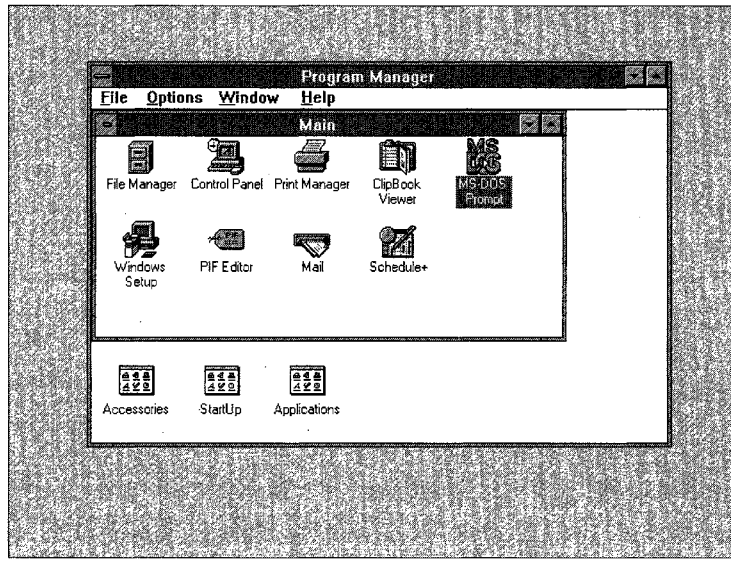


Figure 5-1.
The initial default user screen for Windows 3.1.

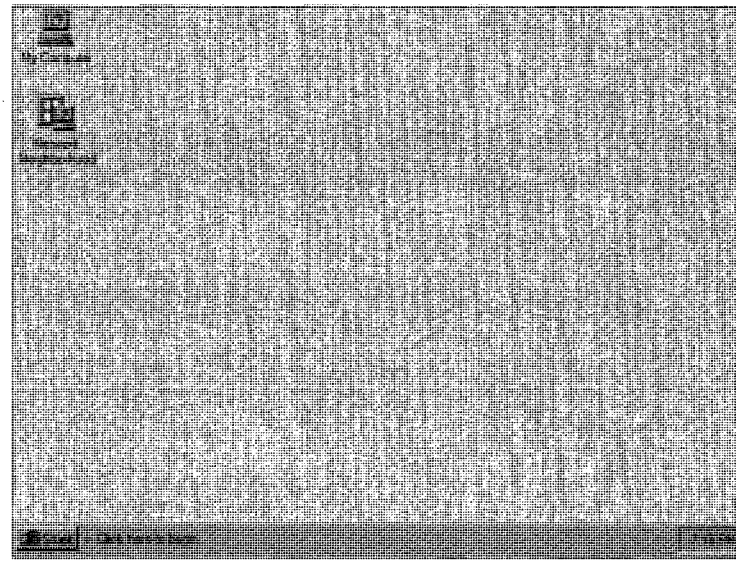


Figure 5-2.
The initial default user screen for Windows 95.

If you're familiar with the UNIX or the OS/2 operating system or with any one of the many products available for MS-DOS or Windows, the term *shell* is no doubt also familiar to you. Generally speaking, the shell is a program that provides the user with a means of control over the system. The shell is the program the user generally considers to be "the system." In MS-DOS, both COMMAND.COM and the MS-DOS Shell provide user control and the system interface. In Windows 3.1, it's hard to point to "the shell." The Program Manager fulfills some of the shell function and the File Manager some more. Neither provides all of the functions that the sophisticated user has come to expect of a good shell program. The new shell is the component that realizes a lot of the user interface improvements in Windows 95. The success of the shell, as the average user's means of controlling the system, will by and large indicate the success of the user interface improvements in Windows 95.

Given that Microsoft rarely alters a successful product simply for the sake of change, you can conclude that there were good reasons for the extensive revision of the interface in Windows 95. One was the desire to take a step toward a fully *document-centric* interface, one in which users concern themselves only with their documents and not with files, programs, directories, disk volumes, and the other odd paraphernalia of operating systems. Microsoft's work on OLE technology laid the foundation for a lot of the thinking that went into Windows 95 and also into Cairo. Windows 95 doesn't quite reach the goal of being a completely document-centric system, but it is a major step forward. It's up to the Cairo team to pull off the final jump.

The other major reason for revising the Windows 3.0 and 3.1 interface was to fix some of its problems—problems either that Microsoft knew about from the beginning or that had become apparent as more and more people began to use Windows. The goal of making Windows 95 easy for users and the desire to attract more new users to the Windows platform warranted a major effort to eliminate these problems.

We'll return to document-centric thinking a little later. Let's take a look at the perceived problems in Windows 3.0 and 3.1 first.

Improving on Windows 3.0 and 3.1

Criticism of Windows became a popular sport shortly after the success of Windows 3.0 began to pick up speed. The continued success of Windows has muted many of the more strident critics, but some critics

made valid points that Microsoft paid close attention to. Within the company, the extensive degree to which Windows served as an application platform created a lot of requests for modification or enhancement of the product. As most reviewers are quick to point out, Microsoft's first release of a product is rarely perfect. But Microsoft does strive to get things right, and most of its products improve dramatically from one release to the next. Windows is no exception, and regardless of whether you consider Windows 95 to be the third or the eighth release of Windows, it does include some major improvements to the user interface.

Windows 95 benefits from the effort invested in the following:

- More unified configuration and control of the system. The plethora of manager programs and other control functions is reduced.
- Improved consistency of the user interface. Similar functions look and feel the same.
- Improved visual details.

System Configuration and Control

Of all the criticisms of Windows 3.0 and 3.1, the most frequent one concerns the confusing variety of managers and control functions.

Program Manager, File Manager, Task Manager

The Windows Program Manager plays a notoriously inconsistent role as a tool for controlling the system. Windows 3.0 and 3.1 include both a Program Manager and a File Manager. The fact that the two different managers allow the manipulation of, in some cases, the same items compounds the confusion many users experience over the relationship between the items displayed in one and in the other. A novice user finds it difficult to grasp the concept of an application program and its separation from data. Even the expert user, for whom the distinction between application programs and files is a known, gets frustrated with the primitive methods Windows 3.0 and 3.1 provide to form an association between applications and documents.[2] Here are a few instances of the shortcomings and inconsistencies in the standard Windows 3.1 managers:

2. Several Program Manager replacement products, such as Symantec's Norton Desktop for Windows, have been very successful by virtue of carefully papering over some of these cracks in the Windows veneer.

- Double-clicking on a filename in the File Manager will start the associated application only if the user (or an installation program) has specifically listed an association between a filename extension and a particular application. If no association has been defined, getting at your data means first running an application and then loading the appropriate data file. This involves a number of steps and a number of names to know or locate.
- The initial Windows desktop shown in Figure 5-1 offers the user no clue as to how to begin working. It displays a confusing collection of icons and names and offers the naive user very little help.
- Application icons can appear on the desktop (the background screen) only when they're running. Otherwise, the icons must reside in one of the Program Manager windows.
- Using the Program Manager to delete the icon that refers to an application or a file is a traumatic experience for many users. The fact that only the icon and the reference to the file get removed is not well understood.
- Similarly, the true meaning of Move and Copy operations for program icons is obscure.
- Filenames composed of 8.3 character strings, with some characters having assigned meanings, are completely inadequate for virtually all users.

The other major deficiency of the Windows 3.1 Program Manager is that it really isn't even a complete program manager. The Task Manager provides some control over running programs. Unfortunately, the Task Manager is confusingly implemented and provides the user with very little actual control over the system. See how many Windows users you know who routinely double-click on their desktop wallpaper to bring up the Task Manager and its list of running applications.

Although it may not happen to you, most Windows users routinely lose windows on their desktops. Because application windows obscure others, a user tends to start the same application twice—thinking that the first instance somehow failed or stopped running. Or the user may believe

that his or her document is completely and irretrievably lost. The Program Manager itself can disappear, causing further consternation. The obscure nature of the Task Manager and of the method for switching between full screen windows compounds the inadequacy of the Program Manager as a mechanism for fully managing every program regardless of its current state.

Control Functions

Although the Control Panel program incorporates most of the components used to effect setup, configuration, and control of a particular Windows system, several other system control functions are hidden away in other corners. Perhaps the best-known example is printer control. Windows 3.1 includes a printer control function in the Control Panel program and an entirely separate Print Manager program. And most applications include a printer setup function accessible from their menu bars. Exactly when to use which control function, and what the results will be, remains something of a mystery even to experienced Windows users. Windows 95 tries to reduce the proliferation of control functions, locating all of them in only two places: one for printer control functions and the other for all other control functions.

Consistency

Another aspect of Windows 3.1 that is treated inconsistently is the particular properties of a control or configuration object. The definitions of how particular items are set up or of how they will respond in certain situations are inconsistent. For example, Windows 3.1 allows you to get to the printer setup option either by choosing Printer Setup in the Print Manager Options menu (see Figure 5-3) or by choosing the Printers icon in the Control Panel (see Figure 5-4).

Both routes lead to the same dialog (see Figure 5-5 on page 164), but neither could be described as a swift or direct route to the most pertinent information. Windows 95 introduces the concept of *property sheets*—a feature aimed at resolving this problem. We'll look at property sheets in some detail later in this chapter.

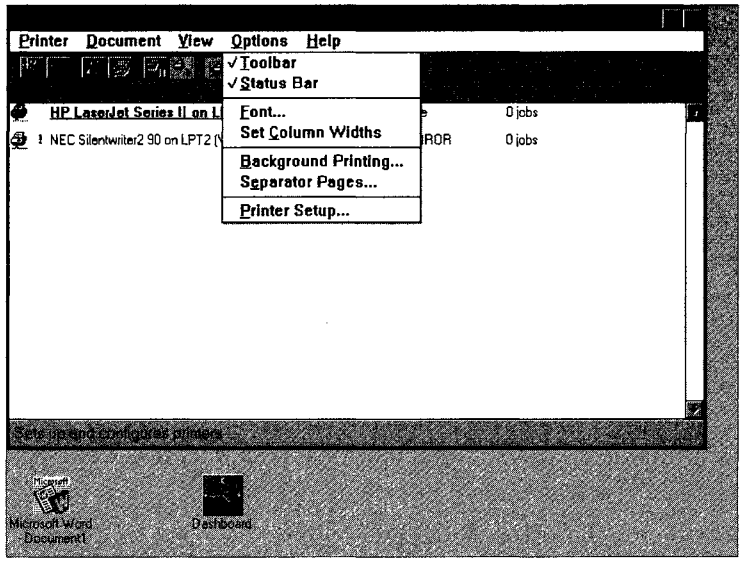


Figure 5-3.
Getting to Printer Setup via the Print Manager in Windows 3.1.

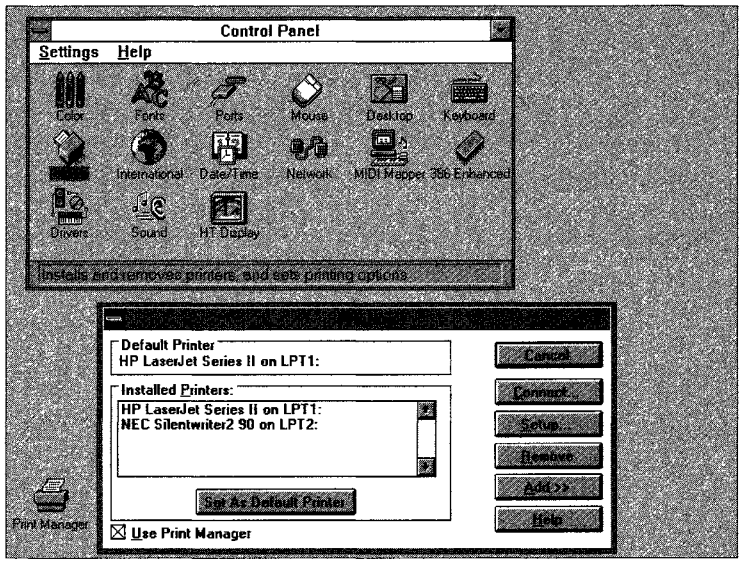


Figure 5-4.
Getting to Printer Setup via the Control Panel in Windows 3.1.

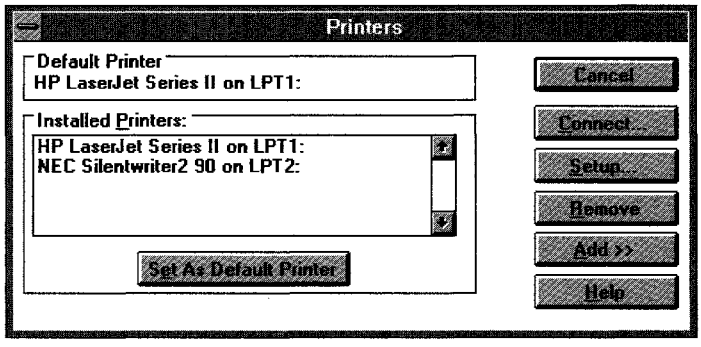


Figure 5-5.
Windows 3.1 printer setup dialog box.

Visuals

The appearance issues the Windows 95 team addressed are minor when you take them up individually. But by carefully eliminating all of the perceived problems and improving the visuals, the team improved the look and feel of Windows dramatically. Essentially, each change amounts to a great deal of attention devoted to every visual detail of the interface. In particular, the team took care to improve the consistency of the screen display and to reduce visual clutter. Take a look at the dialog box from Windows 3.1 in Figure 5-6. Notice that the different controls and buttons are all different sizes and differently aligned. Look at the Screen Saver and Wallpaper groups of controls. In one, the drop-down list box has the arrow button firmly attached to the text box. In the other, the arrow button stands alone. Does this difference have any significance? Actually it does, but this particular visual cue doesn't really help the user at all. The Windows 95 designers were intent on removing such small discrepancies.

Scalability

One other visual design issue also received attention: allowing the user interface to scale better on different display hardware. If you've ever seen Windows 3.1 on a large, high-resolution monitor, you'll have seen that a number of the visual elements don't scale up very well. The system font is one example. With higher-resolution displays becoming more commonplace on popular systems, Windows 95 had to do a better job.

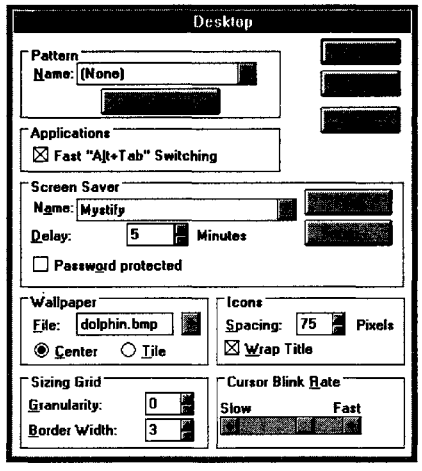


Figure 5-6.
Windows 3.1 desktop control dialog box.

Concepts Guiding the New User Interface

Many of the new user interface ideas for Windows 95 came from the visual design group at Microsoft. These are the people who define, refine, and improve the user interface for all of Microsoft's products. Over the last few years, Microsoft has used more and more visual design expertise on its projects, and Windows 95 is perhaps the first product in which the efforts of the visual design group have had a high level of impact on the appearance and operation of the product. Involved in more than pure visual design, the group works with the development team to define how a product is to respond to user actions. Their goal is to get all of Microsoft's products appearing and behaving in similar, obvious ways. If you know how to use one product, your learning time for another should be greatly reduced. Among other influences, the visual design group uses real people to test hypotheses about interface design—the input often coming from controlled usability testing. Does the user actually respond the way you think he or she should? If not, why not? One team goal for the revised interface in Windows 95 was to reduce the level of knowledge a novice needed in order to begin using the system. The usability tests helped validate whether the design innovations really did accomplish that goal.

In Chapter One, we looked briefly at Microsoft's other major operating system effort—the Cairo project. The initial design for the Windows 95 shell and for many of its interface elements was done by the Cairo group. Throughout the Windows 95 development project, there was a lot of interaction between the Windows 95 and Cairo groups to ensure the consistency of Windows 95 with the evolving Cairo design.[3]

The other major influence on Microsoft's operating system design efforts during 1992 and 1993 was OLE technology. OLE was originally developed by Microsoft's Applications Division as a way of providing a consistent basis for complex data interchange and other application interaction features. OLE rapidly became a more and more important component of Microsoft's evolving software architecture, and in the late fall of 1993, the OLE group moved from the Applications Division to the Systems Division—a move that confirmed OLE's central role in Microsoft's plans. In many ways, OLE can be viewed as the first implementation of Cairo's design concepts. The Windows 95 shell and user interface would be the next major step. Central to all of this work was the evolution of the user interface to a document-centric model, replacing the application-centric view implemented in Windows 3.1.

The Document-Centric Interface

The document-centric interface is the main theme of much of the conceptual work for OLE, Windows 95, and in the future, Cairo. The document-centric approach is derived from the object-oriented concepts that are now increasingly popular in the software industry. Unfortunately, *object orientation* has become an overused marketing term. There are real examples of its use, as in Next's NextStep system, but the proponents of many a system claim that theirs is an object-oriented approach without really implementing one. OLE and Windows 95 are major steps toward a full object-oriented system, although neither of them is complete in that regard. Microsoft intends that Cairo will be.

A document-centric approach means that the users concern themselves only with documents and not with programs and files. The system itself is responsible for maintaining the relationship between data of a particular format and the application that can manipulate the data. Putting the responsibility on the system ties in with the usability information

3. And, of course, one thing the Cairo group did not want was for Windows 95 to appear with features that Cairo would not or could not be compatible with.

that Microsoft has gathered from users of Windows. Many users, particularly those introduced to the PC via Windows, not MS-DOS, find it difficult to separate the concepts of programs and of files. To these users, the item of concern is the document they work on—whether it be a letter composed with a word processing application, or a chart of recent sales results prepared with a spreadsheet application. For many people, the application program and the file containing the specific data are conceptually indivisible.

The document-centric approach contrasts with the approach implemented in most systems today, including Windows 3.1. Today you use an application-centric model. To carry out some operation—for example, redrawing a sales graph in light of the latest month's results—you must first run the appropriate application, then load the data file, then change the numbers, and then redraw the chart. If you want to include the chart in a report, you also have to know how to run the application that handles your report and then cut and paste the chart from its native application into the report file.

OLE introduced the concept of a *compound document.* With OLE, many different types of data can be held and edited within a single document. Editing one element of the document involves simply double-clicking on the object. The application appropriate for manipulating that type of data is loaded without any further action from the user. You see and work with only a single document but possibly several different application programs.

The Windows 95 shell provides a document-centric approach to the system. Everything that can be conceptualized as a document has been. Collections of documents form *folders* (just like file folders), and you can organize folders and documents just as you would organize them in a real filing cabinet.

Look and Feel

The designers and developers of any graphical user interface, such as the Windows GUI, speak of the *look and feel* of the interface. This term refers to two aspects of the interface: the visual appearance of the interface and the behavior of the interface in response to a user action such as a mouse click or a keypress. The appearance and the behavior of the interface are closely intertwined. Many user actions are the direct result of a visual cue. A user who is unfamiliar with the details of a particular operation will seek visual guidance while navigating through a sequence of actions aimed at producing the desired result. Windows, and other

graphical interface products, tend to reduce the learning task associated with a new application by presenting access to many standard operations in the same way. For example, opening a data file within a Windows application always requires clicking on the File menu and then on the Open option on that menu.

Designers of these graphical interfaces worry constantly about a few very important characteristics, asking themselves whether the interface can be described in these ways:

Consistent. Does the user always do the same thing in the same way? Does the user gain access to similar operations using the same keyboard or mouse inputs, guided by similar visual cues?

Usable. Does the interface allow the user to do simple things simply and complex things within a reasonable number of operations? Forcing the user to go through awkward or obscure input sequences leads to frustration and ineffective use of the system.

Learnable. Is every operation simple enough to be remembered easily? What the user learns by mastering one operation should be transferable to other operations.

Intuitive. Is the interface so obvious that no training or documentation is necessary for the user to make full use of it? This aspect of a GUI is the holy grail for interface designers.

Extensible. As hardware gets better or faster—for example, as common screen displays achieve higher resolution or new pointing devices appear—can the interface grow to accommodate them? Similarly, as new application categories become popular, does the user interface remain valid?

Attractive. Does the screen look good? An ugly or overpopulated screen will deter the user and reduce the overall effectiveness of the interface.[4]

In Windows 95, Microsoft addresses many of the issues involved in ensuring compliance with the guidelines set down in *The Windows*

4. Judging by the sales of screen saver software and the semiunderground proliferation of Windows wallpaper and icons, we might conclude that the average computer user is fairly keen on the entertainment value of the interface as well. Designers might not admit to spending a lot of time on this aspect of the interface, but Microsoft introduced a plan to include animated desktops in Windows 95 quite late in the project. Obviously, the Windows 95 designers believed in the value of entertainment.

Interface: An Application Design Guide (Microsoft Press, 1992). This book describes how the appearance and behavior of a Windows application ought to leverage the user's earlier learning. Microsoft is always at pains to point out that the book provides guidelines, not absolute rules. If someone comes up with a better or simpler way to provide a feature, as far as Microsoft is concerned it's fine to go ahead and use it.[5]

The Windows 95 Shell

A lot of design and development effort has gone into the new shell for Windows. During development, one of the major shell functions was referred to by the name Explorer. Whether this name will be used in any form when Windows 95 ships is unknown, but as of mid-1994, the term Explore still appeared on the shell's Start menu. The name does embody one important aspect of the shell's function. The Windows 95 shell is intended to be the program you'll use to explore the system—not just your own desktop system, but also the network system you're connected to. The Windows 95 shell replaces the Windows 3.1 manager programs such as the Program Manager, the File Manager, the Task Manager, and the Print Manager. The Windows 95 shell consolidates the manager functions into a single program that is always accessible and, at least by intent, will be the means by which most users will view and use a Windows 95 system.

One of the more popular terms in Microsoft's Windows group in recent times has been *browsing*. Sometimes it sounded as though all anyone ever wanted to be able to do was to browse around a network, locating files, programs, printers, and whatever. It began to seem as though actually doing something with one of these resources was incidental. That's stretching the truth a little, but Microsoft does intend the Windows 95 shell to make browsing (and thus resource locating) an easy and natural operation.[6] If you study your own work patterns, you'll see that you do spend a significant amount of time locating objects: finding old documents in a word processor directory, for example, or removing old unwanted files to free up disk space. Both of these tasks

5. One new standardized element of Windows 95 is the application tool bar, an interface element used by several early application developers and subsequently copied very widely. The tool bar is a good idea that has become popular with users, so Microsoft decided to include it as a standard element of the Windows 95 interface.

6. Cairo will take this capability much further by providing a powerful query mechanism that will allow the user to rapidly locate any object, anywhere on the network.

involve browsing operations, and improving the efficiency of browsing is a definite positive.

Folders and Shortcuts in the Windows 95 Shell

The Windows 95 shell implements two new concepts that need immediate introduction: *folders* and *shortcuts*.[7] Folders are a foundation of the shell design, and as you use Windows 95, you'll quickly find that shortcuts are a valuable enhancement. A lot of the examples in the upcoming pages will display the use of folders and shortcuts to one degree or another. We'll take a look at shortcuts in the next section.

Folders A number of folders and their contents are shown in Figure 5-7. A folder is a logical container that allows you to group any collection of items you choose—a set of documents produced with your word processor, for example. The items, or objects, a folder can hold include

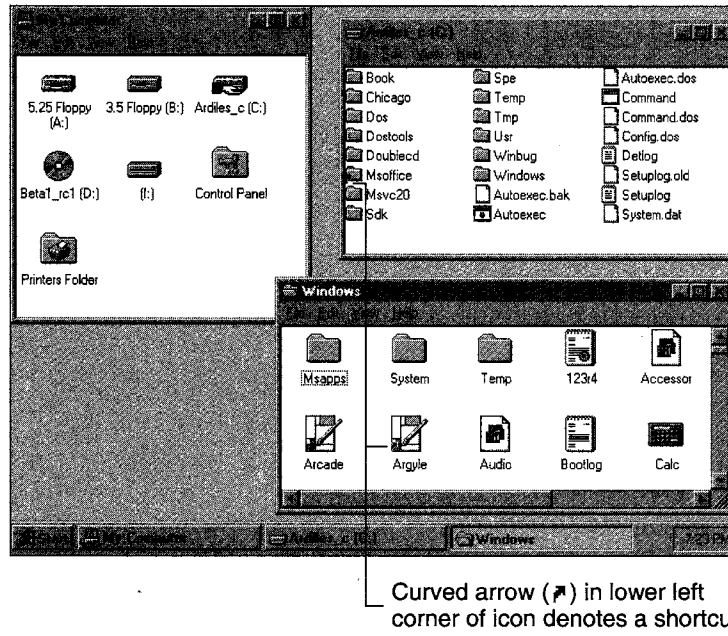


Figure 5-7.
Folders in the Windows 95 shell.

7. Microsoft originally used the name "link" to refer to this feature. As expected, it did change before product release. Among other candidate names, "nickname," "remote control," "jump," and "post it" were under consideration. The term "shortcut" was chosen in early 1994. Whether it will be the final term remains to be seen.

individual files, other folders, or shortcuts. (Notice the curved arrow mark used to visually denote a shortcut.)

The shell provides a view of both the local and the network system that is an exact replica of the filesystem—that is, an object shown in one of the shell's windows is actually a file or a directory residing on a disk somewhere. Folders are directories, and even shortcuts are stored as files. This design is different from that of other implementations in which some objects really are files and others exist in another universe. In Windows 3.1, for example, the icons in the Program Manager groups exist physically either as individual files or as resources within executable files; entries in a .GRP file in the \WINDOWS directory link the icons to the program groups. When you try to track down the icons outside the Program Manager, you need special knowledge to do so. Windows 95 makes everything a file or a directory, so most special files (such as the .GRP files) disappear. If you know how your desktop looks, you know how your files are organized, and vice versa.

The generalized folder mechanism, with its ability to contain any other object, is a big step on the way to a completely document-centric system. Operations such as printing, copying, and searching through a document require no knowledge of the particular program used to implement the operation. Any operation is available in a completely general way for any document. And one of the most important design goals for the shell is to provide a fully consistent environment. An operation on one kind of object achieves predictable results based on what you know about the behavior of the same operation with a different kind of object. The use of the folder concept is key to achieving this consistency.

Shortcuts The Windows 95 shortcut concept is a very powerful one. It allows you to create a reference to an object without having to make a copy of the object. For example, you might create a folder containing several word processing documents together with a shortcut to the printer you use for output. Figure 5-8 on the next page is an example of how this folder might appear. To print a document, you'd simply open up the folder, click on a document icon, drag the icon to the printer icon, and drop it. Access to the appropriate printer would be immediate, and the document would be printed without your needing to specifically run the application you used to create the document. The shell would take care of loading the appropriate application and informing it of the operation (printing) and the chosen document.

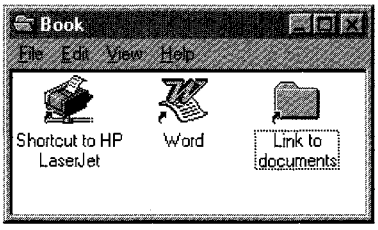


Figure 5-8.
Shortcuts in the Windows 95 shell.

Windows 95 uses shortcuts extensively, and you'll see several other examples of their power in this chapter.[8] Although Windows 95 continues the use of a hierarchically organized filesystem, the availability of the shortcut mechanism makes it possible for you to organize your documents the way you want them, without having to make multiple copies of particular files or programs. For example, if you keep several folders of documents that require the use of a calculator while you're working on them, you can store a shortcut to the calculator in each folder. The calculator is then immediately accessible, and you don't have to make multiple copies of the calculator program. Although purists might frown at the ability of shortcuts to muddle a pure hierarchical filesystem structure, usability tests have shown that very few people are comfortable with the constraints of a strict hierarchy. People don't work hierarchically, and they dislike the hierarchical filesystem for forcing them to try to.

Windows 95 implements shortcuts in the shell by recording their existence in a .LNK file. Each shell folder that contains shortcuts, and thus each disk directory associated with a folder, contains a .LNK file for each shortcut.[9]

Desktop Folders

Desktop folders in Windows 95 are very dynamic, and thus the contents of the associated disk directories change frequently. A \DESKTOP directory on the system's boot drive contains all the items that define

8. Something akin to links is in use in the Windows 3.1 Program Manager: icons in program groups are links to the executable program. Other desktop utilities extend the capability. However, Windows 3.1 neither formalized nor generalized the link concept.

9. Originally shortcut information was stored in a DESKTOP.INI file that also held window placement information for shell folders. DESKTOP.INI eventually disappeared in favor of directories collected under the Windows \DESKTOP directory.

the initial layout of the user's desktop. As items are moved on and off the desktop, the physical contents of the \DESKTOP directory change. Figure 5-9 shows a desktop layout and a listing of the associated disk files that track this configuration. Notice the default SHORTCUT.LNK files that contain the shortcuts to the printer object.

Figure 5-9.
Desktop folders in the Windows 95 shell.

System Setup

System setup in Windows 95 is considerably improved over Windows 3.1 setup. As part of the overall goal to make the system easy to use, system setup makes it simple for the new Windows user to install the system and get it running for the first time. If you know what you're doing, you can still customize your system as you install it. But if Windows is a new adventure for you, the answers to a few simple questions are sufficient to get you going. Microsoft's Plug and Play technology is central to the improved setup process.

Microsoft's usability tests uncovered the difficulty new Windows users had with getting the system to do something—anything—the very first time they tried to use it. In retrospect, it's perhaps easy to see why. Look at

the Windows 3.1 screen display in Figure 5-1 back on page 158. Nothing on the screen provides a hint about how to start—and the StartUp icon can even mislead. There's a lot of information, but no discernible first action. The problem is compounded by the physical difficulty many beginning users have with the mouse double-click action. In Windows 3.1, unless you can double-click after installation, it's very hard to get the system to do anything for you. This isn't a problem limited to Windows. Most graphical systems today still require users to possess quite a lot of information and skills before they can start to use the system.

Microsoft addresses these problems early on in Windows 95. The single "Start" button on the screen (see Figure 5-2 back on page 158) is a good hint. To make sure that the user doesn't miss the Start button, the status message alongside bounces against the button when the user first starts the system—like a finger pointing to the correct path. As the user continues to work with the system, other helpful hints appear as status messages.

The Initial Desktop

With the initial default desktop in Windows 95 (see Figure 5-2), there is but a single obvious point of access to the system—the "Start" button in the lower left corner. The area at the bottom of the screen is called the *system taskbar.*[10] In the initial configuration, the empty desktop and the message on the taskbar telling you exactly what to do leave you with only one real choice. In fact, double-clicking on the desktop computer icons also gets the user going. Clicking on the "Start" button will get the user to the screen shown in Figure 5-10. Selecting any items with continuation menus offers yet more possibilities. Figure 5-11 shows one of these possibilities.

To get this far, the user must at least have mastered the single-click operation with the mouse. Simply moving the mouse to one of the items shown on the menu in Figure 5-10 means that you're almost home. One more click, and you're running an application. Microsoft believes that this simplified setup and first time operation of the system will quickly get users to the point at which they're doing real work, rather than fooling around with the system. It's hard to come up with a

10. Another term that may yet change. "Tray" was the term used for a long time.

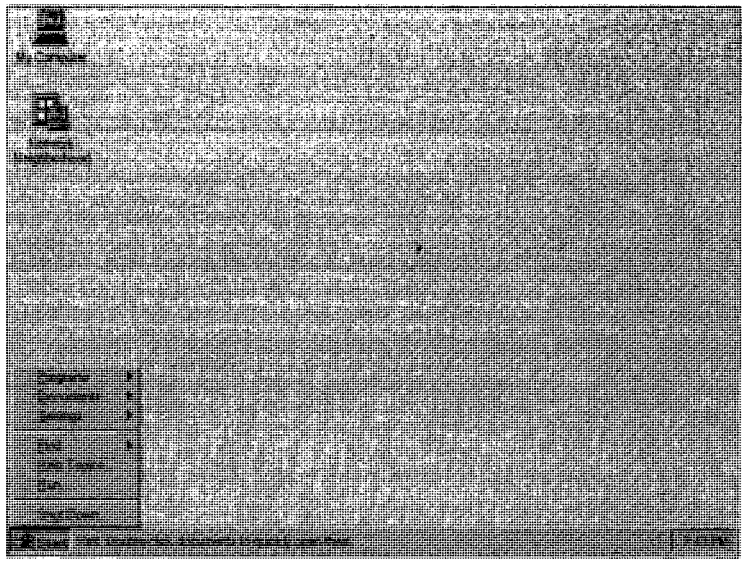


Figure 5-10.
The default Start menu.

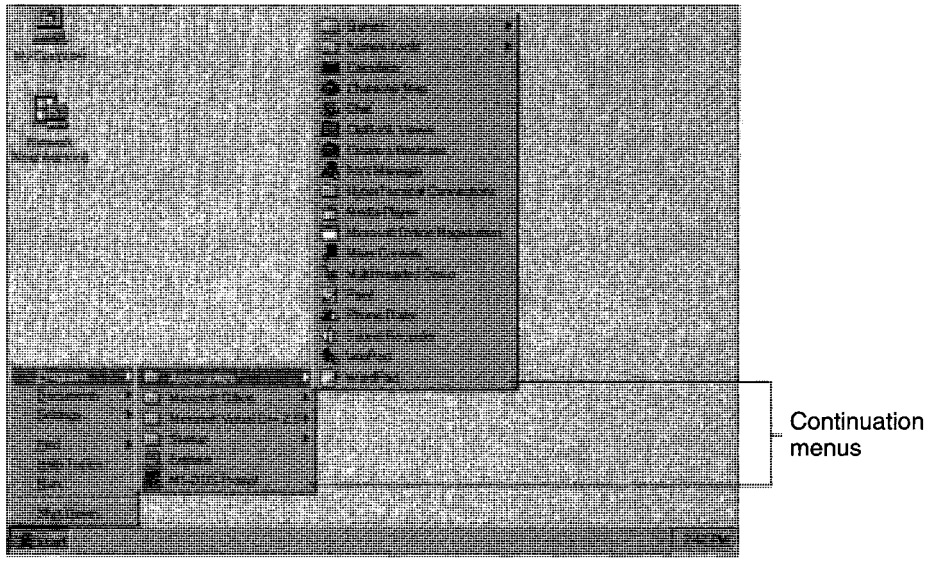


Figure 5-11.
Continuation menus.

general purpose scheme that's faster than two prompted mouse clicks from startup to application, so the expectation appears to be justified.[11]

The only other access points on the initial desktop are "My Computer" (which the user will promptly rename) and "Network Neighborhood" (which appears only if Setup detected a network connection). Figure 5-12 shows these folders after double-clicking has opened them. The user can explore the local system further by double-clicking on the disk icons and can explore the network by double-clicking on the other systems that are active.[12]

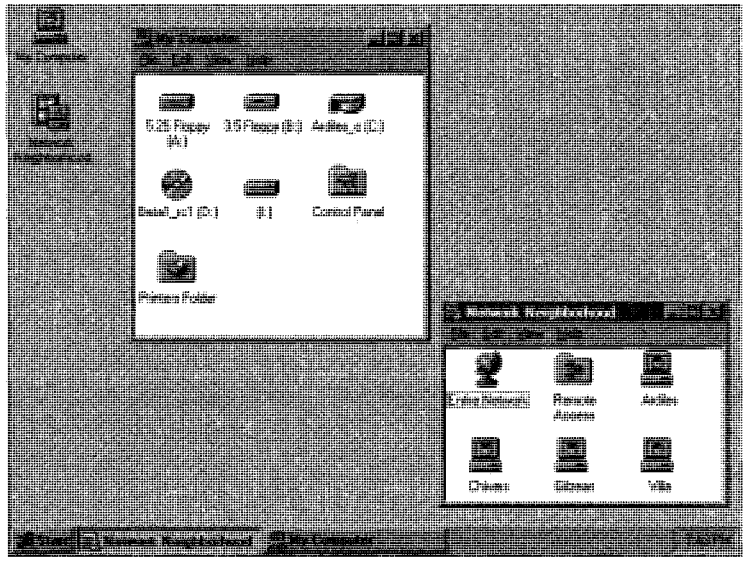


Figure 5-12.
Other access points on the initial desktop.

11. If you think this is an at all unreasonable amount of effort to get users to the point of running an application, Microsoft's usability testers have some videotapes for you. The tapes show novice users taking several minutes (and in some cases giving up the attempt) to locate and run Notepad under Windows 3.1. In the same test under Windows 95, the time was reduced substantially.

12. Early versions of the shell allowed access to the entire network from this point. On a large network (such as the Chicago development group's), accessing the entire network produced a lengthy and nonuseful list of network resources. The neighborhood concept allows you to constrain the network resources you view to the resources you're interested in.

The Desktop

In Windows 95, a number of new design ideas underlie the new look and behavior of the desktop. In Windows 3.1, the user's conceptual desktop consisted of the Program Manager and its program groups and to some extent the background. Beyond holding minimized windows and providing a display area for the user's favorite screen wallpaper, the background didn't do much. Windows 95 changes that significantly. The Program Manager is gone, and the background becomes an important part of the overall shell design.

On the desktop, Windows 95 implements a look and feel that is consistent across all objects. Drag and drop operations are supported everywhere. You can move folders by means of drag and drop operations, and as we've already noted, you can print documents by dragging them to the printer and dropping them. The screen background itself becomes an integrated part of the desktop. You can drop objects on the desktop for storage. You can create storage objects and put them on the desktop for safekeeping. Conceptually, the Windows 95 desktop is intended to serve just as your own real desk in your own real office does—even to the extent of allowing you to put pictures of the family dog on it.

As you gain experience, your desktop will probably look something like those shown in Figures 5-13 and 5-14 on the next page after you've been using Windows 95 for a while. The desktop itself acts as a storage medium for any objects you put there: folders, shortcuts to objects, and additional access points to the system such as the local system and the network. Some of the icons will probably appear on every desktop because they represent specific points of access to the system. Other objects on the desktop will reflect the user's personal customization of his or her working environment.

The computer icon provides access to your local disk storage. Your opening this object is intended to convey an impression of your "opening" your computer to inspect the information it contains. Earlier on, we looked at some of the system folders and at some aspects of the shell's facilities for browsing as you deal with folders. The network neighborhood is the point of access to the systems you have connections to. Figure 5-15 on page 179 shows an example of the hierarchy of folders opened across the network as the user looks for a particular file.

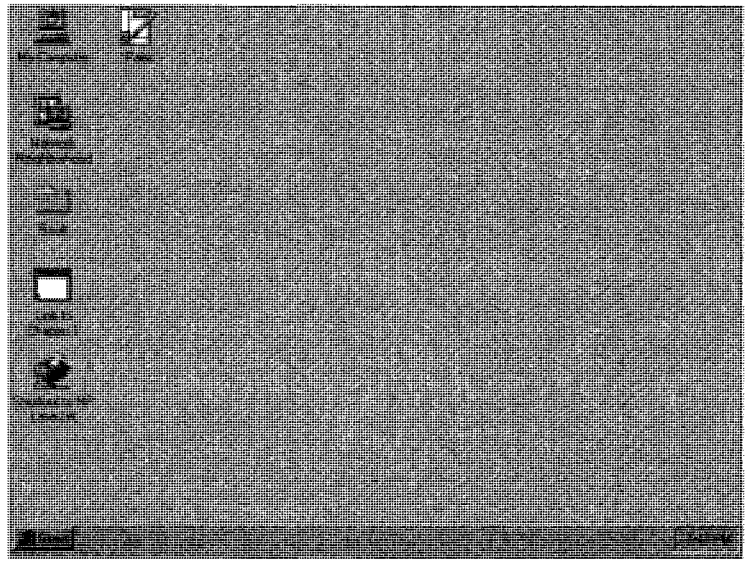


Figure 5-13.
A user's desktop in Windows 95.

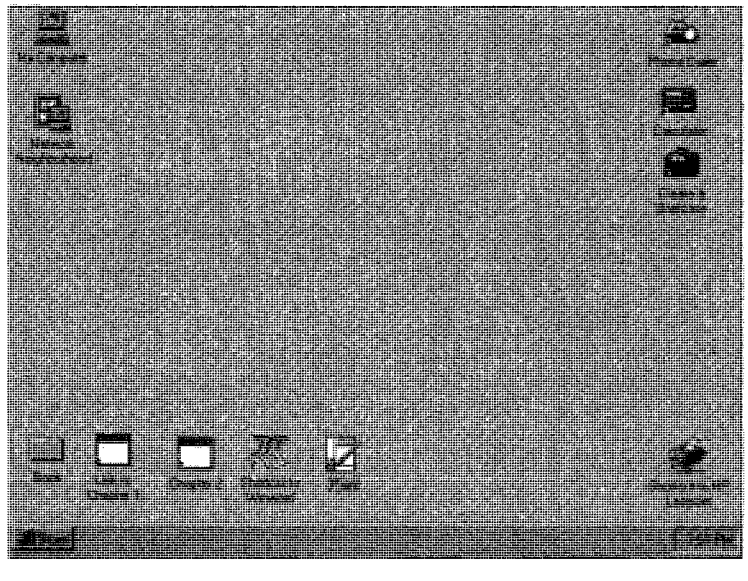


Figure 5-14.
Another user's desktop in Windows 95.

The system taskbar at the base of the screen represents a permanently available "home base," or anchor point, for the user. By default, the system taskbar is always visible and accessible. The Windows 95 designers intend the system taskbar to keep the novice user from losing his or her place in the system. Even when an application maximizes its window, the taskbar is still visible and the user can access it.

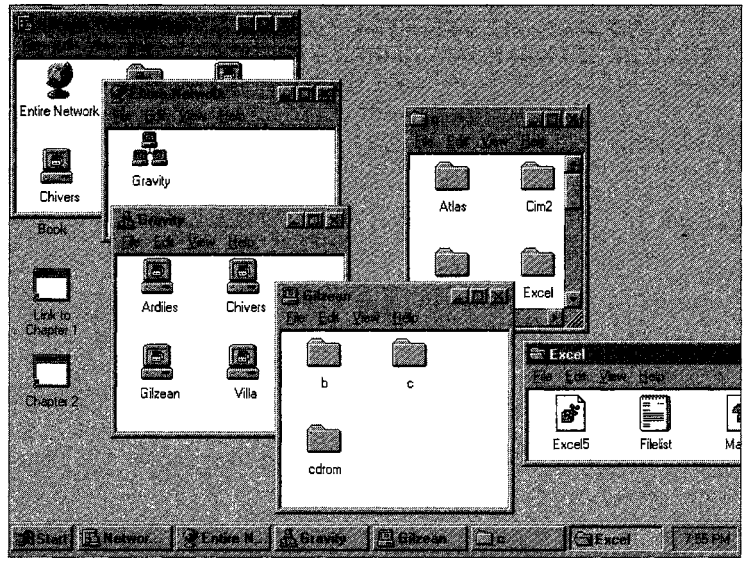


Figure 5-15.
Browsing the network from the desktop.

The Taskbar

Losing windows on the desktop has been an all too common problem with Windows 3.1, notably when minimized windows got hidden behind other windows. To solve this problem, Windows 95 introduces the taskbar—a user interface element that serves as a common storage point for several different types of objects. As you'll have noticed from the earlier figures in this chapter, you'll see the taskbar on the screen nearly all the time. The default taskbar behavior is to always be visible. Windows 95 applications must content themselves with the physical screen dimensions that are left. A maximized window occupies the entire physical display except for the area used by the taskbar. If you turn off the *always on top* property for the taskbar, a maximized window can obscure the taskbar. This is not quite the same behavior as that of contending windows under Windows 3.1. The *always on top* attribute in

Windows 3.1 would cause a window to obscure some of the maximized window underneath. The apparent screen dimensions did not change as they do in Windows 95. Microsoft has added an Auto hide option for the taskbar. Setting this option will cause the taskbar to appear only when you move the mouse cursor to the edge of the screen at which the taskbar rests. The taskbar will disappear when the cursor moves away from that edge of the screen.

In the taskbar, you'll see the following:

- The single button that provides immediate access to some common system functions: help or system shutdown, for example.
- A resting place for active windows. The system will put a button representing each active window into the taskbar. This refuge solves the Windows 3.1 problem in which minimized-window icons disappeared when they were hidden behind other windows.

The user can configure the location and size of the taskbar. Figure 5-16 shows an alternative layout. This particular layout makes it easy to demonstrate the various uses for the taskbar, but it probably isn't one you'd choose because it significantly reduces the screen space left for applications. And the shell does limit the configuration possibilities. You can adjust the size of one dimension of the taskbar, but the taskbar must rest against one physical screen boundary, and its larger dimension is always the same as that of the chosen edge.

One major function of the taskbar is to provide a consistent "home position," or anchor point, for the user. If you accept the taskbar's default behavior, the taskbar is always visible. Then, if you get confused or the desktop gets thoroughly messed up, the taskbar is always there as a place to return to for help or other system functions and to reorient yourself.

Application compatibility issues in relation to the system taskbar are quite interesting. Ultimately, the designers decided to treat the area occupied by the taskbar as if it were off the edge of the screen. Thus, Windows 95 clips the window in Figure 5-16 much as if the user had moved it past the right-hand physical edge of the screen. In Figure 5-17, the application believes it is running in a maximized window and occupying the whole screen. Windows has actually reported the screen dimensions to the application so that it excludes the area used by the taskbar.

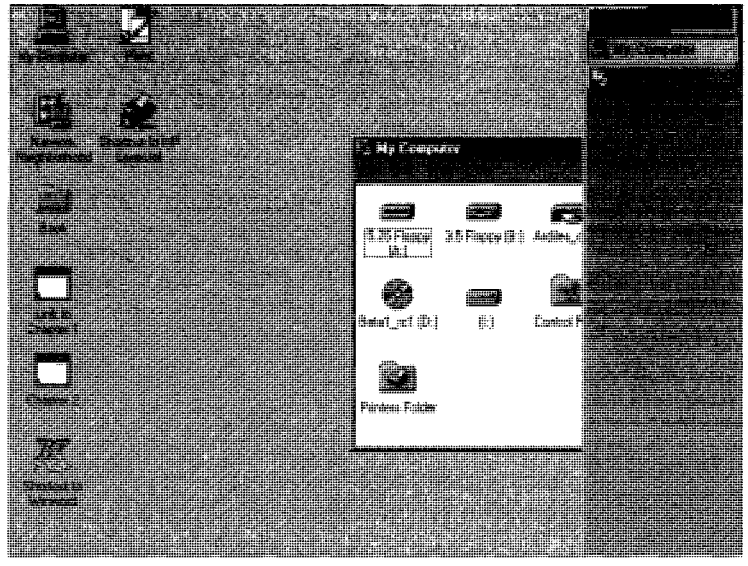


Figure 5-16.
The system taskbar in an alternative layout.

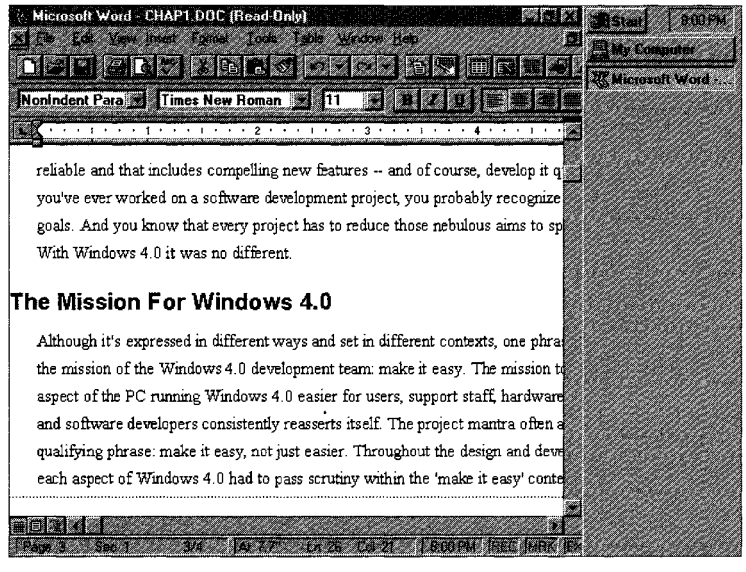


Figure 5-17.
The taskbar and a maximized application window.

On-Screen Appearance

In the example screens we've already looked at, you've no doubt noticed many of the innovations in the on-screen appearance of Windows 95. A lot of effort went into refining the overall appearance of the product. Some changes, such as the introduction of the system taskbar, are obvious, but there are many subtle design changes throughout the product as well. And many specific visual elements have changed in Windows 95. You may have noticed already the changes in the minimize and maximize icons on the application title bar. We'll look at several other changes later in this chapter.

The example screen detail in Figure 5-18 shows some of the subtle aspects of changes in the Windows visual elements. This part of a screen shows a Windows 3.1 application alongside the Windows 95 system taskbar. If you examine the application buttons closely, you can see that the alterations are very slight: in the system taskbar, some of the black outline disappears, and the shading details change. As you look at an individual element, the change doesn't seem very significant. However, when replicated in every element of the Windows 95 interface, this level of detailed change does produce a much softer, more visually pleasing, and consistent appearance. You can see this attention to visual detail throughout Windows 95—a case of the whole amounting to more than just the sum of the parts.

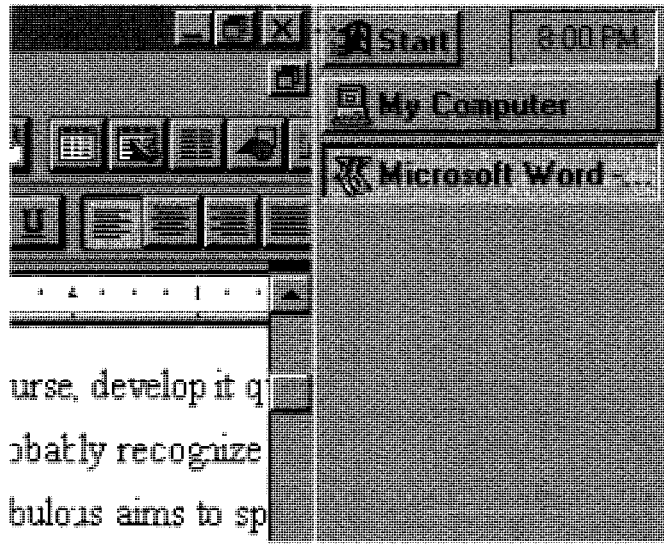


Figure 5-18.
Windows 95 screen detail.

Figure 5-18 also highlights an interesting side effect of the redesign. The buttons on the application's menu bar show the new-style minimize, maximize/restore, and close icons, and their appearance follows the Windows 95 conventions. The application's button bar, on the other hand, retains its "older" style. The button bar wasn't a standard control in Windows 3.1, so the application has to draw its own buttons. Under Windows 95, an unmodified application will continue to do that, whereas the standard controls are drawn by the system itself, so they adopt the new style and appearance.

Another theme in the redesign for Windows 95 is the provision of visual cues to the user as often as possible. In earlier examples, you may have noticed that the minimize and maximize buttons convey the appearance of minimized and maximized windows and that specific application icons are embedded within document and folder icons. Figure 5-19 shows screen detail from a more obvious example, in which the user is examining a disk drive. The type of the drive (the hard disk graphic), the space used in comparison to the available free space (the pie), and the fact that it's a network drive (the connecting cable) are all shown pictorially.

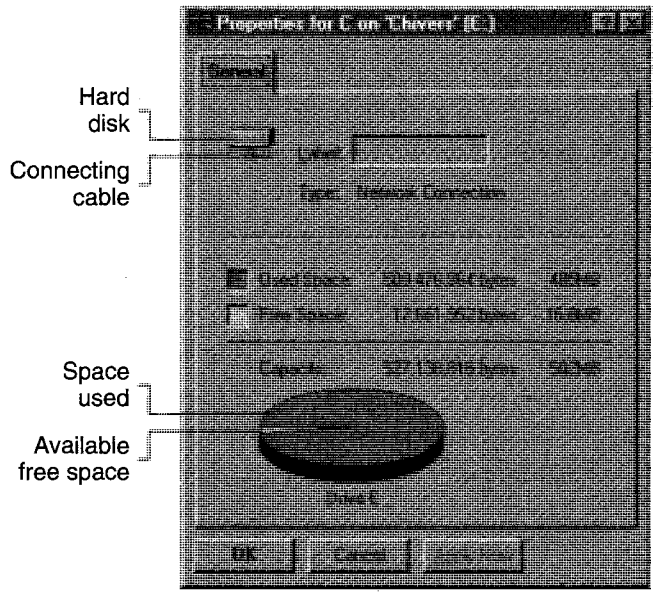


Figure 5-19.
Visual cues in Windows 95.

Light Source

Another theme of the design for all the visual elements of Windows 95 was the adoption of a consistent light source. The imaginary source "shines" from high and wide over your left shoulder as you look at the screen. All the shading for the three-dimensional effects uses the same light source. The screen detail shown in Figure 5-20 demonstrates this consistency. The sunken field containing the LPT1: string, for example, is shaded on the left and upper edges, and the raised New... button is darker on the bottom and right edges. In Windows 3.1, the light source isn't entirely consistent, and you can find examples in which the light "shines" from different places. Again, this is an apparently trivial attention to detail taken in isolation, but it does add a lot of polish and coherence to the product as a whole.

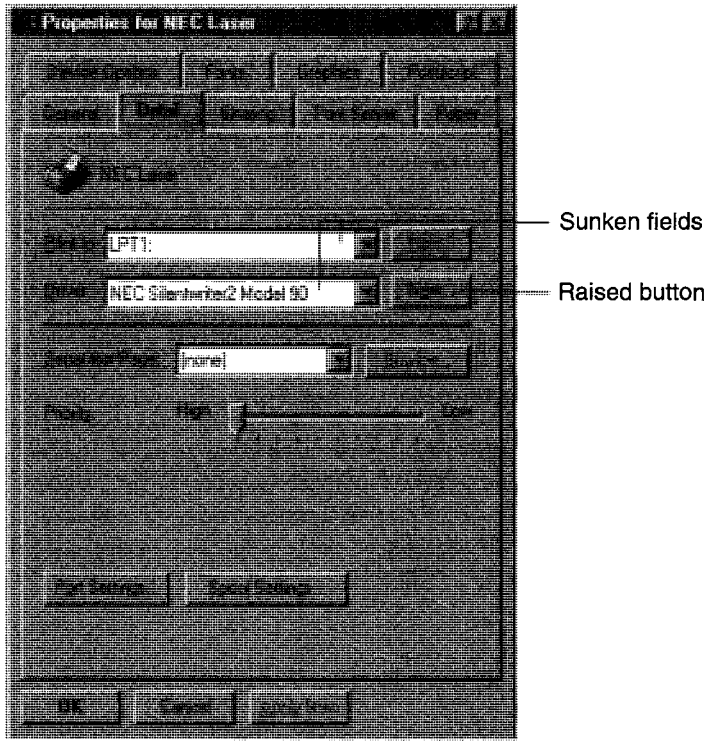


Figure 5-20.
The Windows 95 light source made consistent.

Property Sheets

Windows 95 attempts to introduce a much higher level of consistency for access to object properties by making use of the secondary, or right, mouse button (yes, finally a use for the other button!). Clicking the right mouse button on any object will produce a popup menu that includes a Properties item. Selecting the Properties item leads to a new control called a *property sheet.* A property sheet is similar to a dialog box in many respects and can include checkboxes, buttons, and editable fields—in fact, any kind of control. Within the property sheet lies all the information about the configuration of the selected object. Figure 5-21 on the next page, for example, shows the property sheet for the desktop. Note a few points about objects and property sheets:

- The popup menu for the object appears when you right-click on the object itself.
- An object's property sheet can have multiple pages marked by tabs—much as a book might have its sections separated by tabbed dividers. This provision for multiple pages allows a single property sheet to include a lot of information that doesn't have to be jammed into one enormous dialog box.
- You make page selections in a property sheet by simply clicking on the appropriate tab.
- Consistent with the Windows 95 theme of providing visual cues, the property sheet that controls the monitor configuration provides a representation of the display and its screen appearance, the property sheet for printer configuration provides a representation of a printer, and so on.

The obvious intent is to persuade all application developers to adopt the same conventions with respect to use of the right mouse button and the property sheet control. If that happens, object property inspection and modification will be completely consistent under Windows 95. Windows 3.1 applications won't respond to the right mouse button click or display property sheets since an application must be modified to do so.

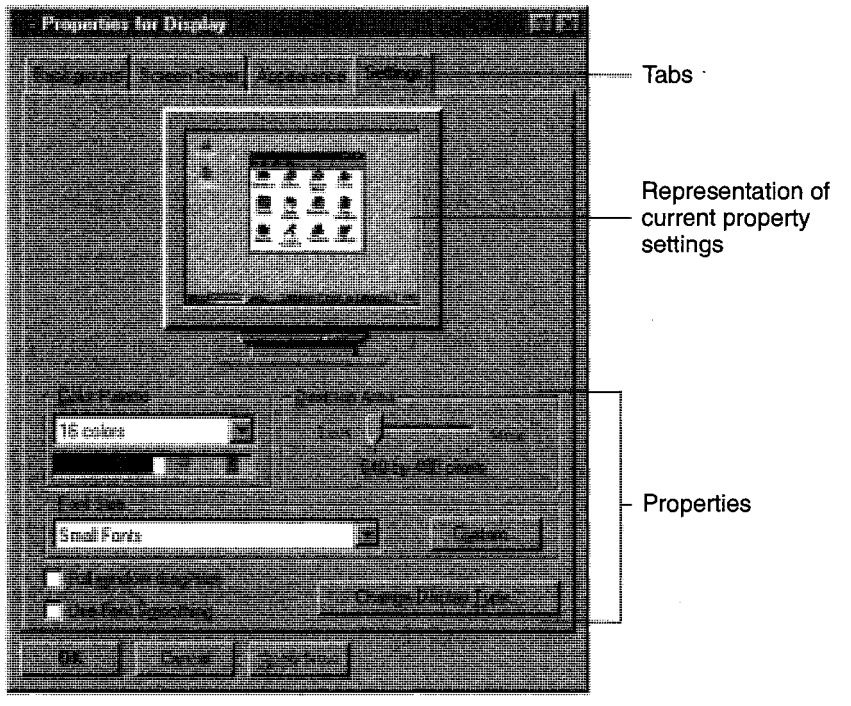


Figure 5-21.
Desktop property sheet in Windows 95.

Online Help

If you've ever tried to find your way to some deep, dark Windows secret, chances are that you found the online help system rather tedious and frustrating to use. You probably found that there was a lot of information to browse through, and you probably had to do a lot of backtracking before you finally unveiled the secret. You weren't the only one. Microsoft's usability studies showed that this was a common problem. Windows 95 adopts a much more direct approach to online help presentation. The help text is shorter, more explicit, and more context sensitive. Microsoft is encouraging application developers to adopt similar guidelines for revisions to online help in application products. The Windows 95 help system is unlikely to be perfect, though. There always has to be a compromise between simple, direct instructions that satisfy 90 percent of the user's needs and lengthier treatments of the more obscure details. No doubt we'll see more improvements in future releases of Windows.

Here are some of the changes to the Windows 95 online help:

- Keeping a persistent access point available to the user. The "Help Topics" item on the Start menu is always available.[13]
- Taking a task-oriented approach to the online help text. The text describes explicitly the steps the user must take to accomplish his or her goal instead of providing a general description of the topic. Figure 5-22, below, and Figure 5-23 on the next page include examples of this new format.
- Making sure the help window remains visible throughout. There's no need to click back and forth between the window you're trying to work with and the obscured help window that describes what you're supposed to do. As you can see in Figure 5-22, the active window is the Find File window but the help window is still visible.[14]

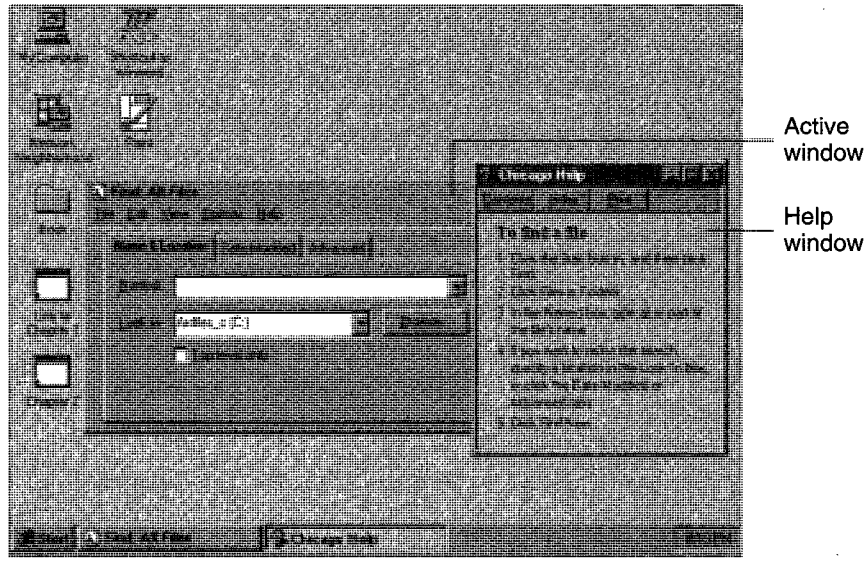


Figure 5-22.
A Windows 95 help window.

13. For a long time, the shell included a help button on the taskbar.

14. This example also points up one of the problems in keeping the help screen visible. The help window obscures the Start button in the Find File window.

- Reducing the help verbiage and the steps you have to take to complete an operation for which you need help. The text is simpler and more direct, and the help windows include shortcut buttons that take you directly to the system function that will complete the operation. Figure 5-23 shows an example. Clicking on the button will immediately display the desktop properties screen saver sheet.

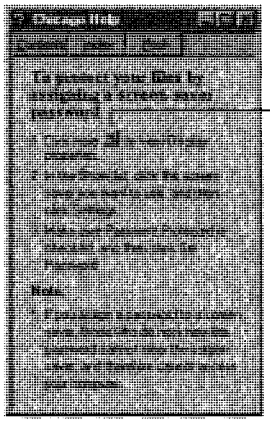


Figure 5-23.
A help shortcut in Windows 95.

- Heightening context sensitivity. The help for an individual field within a dialog box is for that field, for example, and not simply a link to the help text for the entire dialog.

Implementation

Apart from enabling the shell as an OLE client, the Windows 95 team introduced three other features of the shell implementation worth noting:

- 32-bit code. The shell is a 32-bit application that makes full use of the Win32 API.
- Multithreaded processing. The shell takes advantage of the threading capabilities of the system. Each window opened by the shell runs as a separately scheduled thread. You'll see this innovation in action if you move the hourglass mouse cursor outside a window boundary. The cursor will change back to

the normal arrow pointer, and, yes, you can actually continue working, moving to another task.

- Shell extensions. Acknowledging its competitors' desire to extend and improve the Windows interface, Microsoft has included a lower-level interface that allows other vendors to integrate extensions of the basic Windows shell.

Design Retrospective

We've now looked at each of the major new concepts introduced with the Windows 95 shell. Of course, some of the concepts come from much earlier work on user interface design outside Microsoft, and many have evolved from earlier versions of Windows and Microsoft application products. The Microsoft designers didn't simply sit down one day and draw up the design for the Windows 95 shell. During the course of development (and indeed, during the preparation of this book), the design of the shell has changed quite a lot. It's worth looking at how and why these changes came about.

The Outside Influences

Throughout the history of Windows, Microsoft has taken vociferous criticism of the user interface. Some of the criticism is attributable to the product's success, some of it to the detailed legal scrutiny the interface underwent during the long-running dispute with Apple Computer, and a great deal of it to the simple fact that people tend to be opinionated about interface issues. Very few people care a lot about the names of Windows API calls or about the order of parameters passed to a function. But everyone has an opinion about the user interface. So whether they wanted to be or not, the Windows 95 designers were the focus of a lot of attention when they began to show prototypes of the shell.

By the time of Microsoft's first major Windows 95 design review—a meeting in Redmond in July 1993 that hosted about 25 people from the leading PC software development companies—most of the shell's features were in place, ready for the product's first external release. Much animated discussion at this meeting, and much more on the private CompuServe forum that hosted the early testers of Windows 95, helped shape the thinking behind the next release.

Although Microsoft sought and received a lot of expert opinion on the shell's design, one principal influence was the series of usability tests

it conducted throughout 1993 and 1994. In some 30 separate tests involving as many as 12 people at a time, Microsoft observed a mix of users trying to complete tasks using the new shell. The users included people who had never used Windows (although they had used MS-DOS) as well as Windows 3.1 and Macintosh users. Microsoft augmented these tests by interviewing people who trained Windows users.

Among the user difficulties identified by the usability tests, these seemed to consume most of the design thinking during the development of the Windows 95 shell and user interface:

- Window management—dragging windows and sizing them, and the implicit ordering of the windows on the desktop.
- The difference between the windows supported by multiple document interface (MDI) applications and single document interface (SDI) applications. (Try to explain to a novice why the Windows 3.1 Program Manager apparently clips some windows and not others, and you'll see the problem.)[15]
- The concept of hierarchical containment. Experienced computer users have learned to live with hierarchy, but putting a folder in a folder inside another folder is certainly not the way most people organize a filing cabinet.
- The mouse double-click action. If you are innocent of experience and receive no instruction, it's almost impossible to guess that you need to double-click.

The Development of the Shell

The design work for the shell really began back in 1990, although at the time the effort wasn't even thought of as Windows 95 interface design. Later a lot of the Windows 95 shell design work was done in conjunction with the Cairo team's work to ensure long-term consistency between the two products.

These days Microsoft uses Visual Basic to prototype almost every screen display. The shell has been no exception. In addition to the obvious advantage that people can see and show each other what they're

15. MDI vs. SDI was a hot topic during Windows 95 design reviews. Ultimately, the team decided that Windows 95 would be an SDI system because they believed SDI to be easier for users. But since many software developers had invested in it, MDI support would still be there.

talking about, VB prototyping makes it possible to develop an early working model of the design. Although most operations won't have any effect yet, you can put together a prototype sufficiently rich that you can get real users to come and try it out. This kind of prototype is what was used most often in Microsoft's usability tests.

Microsoft released the first external test version of Windows 95 in August 1993. This so called M4 release was a major milestone for the development group since it represented the beginning of the end of the project. The subsequent M5 release was scheduled for the huge Win32 software developers conference Microsoft hosted in Anaheim in December 1993. In between M4 and M5, the shell development team concentrated on transforming the shell from its 16-bit state into a true 32-bit application. The design team in the meantime went back to thinking and usability testing.

Immediately after the M4 release, Microsoft undertook a six-week design project that put members of the Windows 95 and Cairo teams together to refine the shell design in light of current knowledge. This design effort focused largely on

- Learnability—how to get people doing productive work in the shortest possible time
- Usability—how the observed tests should guide refinement of the shell to make common tasks easier than in Windows 3.1
- Safety—how to achieve an environment in which no user should ever have to worry that his or her actions might destroy data
- Appeal—how to get people to like the Windows 95 shell; how to harness the naturally polarized opinions of the users to foster an emotional attachment to the shell

The result was a new prototype presented in an internal design review meeting with Bill Gates in late September 1993.[16] In this meeting, the team introduced the changes to the shell's folder mechanism, a new design involving novice and expert modes of the shell, and animated desktops. As they came out of that meeting, the shell design

16. Such meetings are a standard ingredient of Microsoft's development process. Always approached with much energy and not a little trepidation, a "BillG review" continues to have a significant influence on every Microsoft product.

team believed that pending a final decision on the transfer model (which we'll look at a little later in this chapter) and a host of small details, they were close to a final design. All they had to do, they thought, was wait for the programmers to finish the 32-bit conversion for the M5 release, and they could have the user interface they really wanted. This didn't turn out to be true since the novice and expert modes were later dropped and the detailed operation of the system taskbar underwent further changes.

Changes in the Shell

The biggest change in perspective that took place during the course of the shell development project was seeing that the novice user and the experienced user should be treated differently. Figure 5-24 shows the default startup screen used in the M4 and M5 releases of Windows 95.

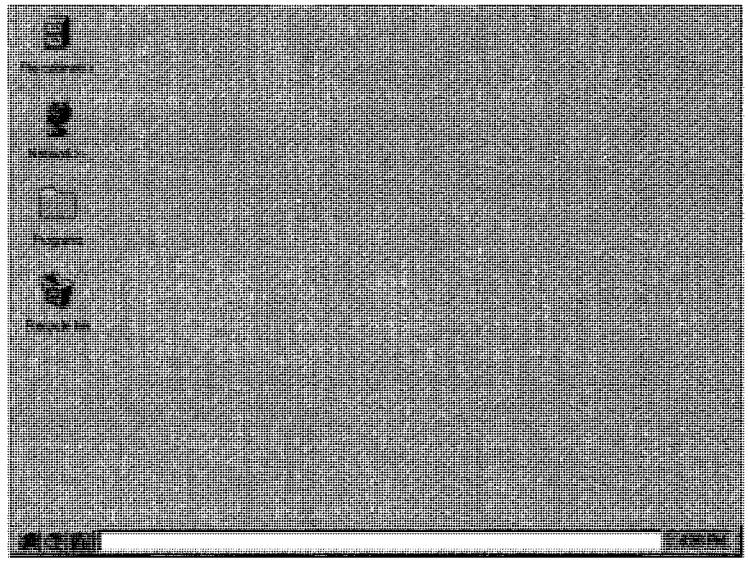


Figure 5-24.
Prototype default startup screen for Windows 95 M4 and M5 releases.

Contrast this prototype with the eventual design we've seen in Figure 5-2 back on page 158, and you can see some big changes:

- The default startup screen in the prototype shown in Figure 5-24 offers several points of access to the system. The taskbar,

for instance, includes three buttons rather than one, and the Network icon, the Programs folder, and the File Cabinet icons on the desktop seem to suggest even more avenues of approach.

- There's no hint to the user about how to begin.

After the M5 release, the design introduced the explicit notion of a novice mode and an expert mode. Users who acknowledged themselves to be novices would see a shell configuration that painstakingly guided them through the system.

Figure 5-25 shows an example of the novice interface. Eventually this separation of users was dropped, and it never was a feature in any of Microsoft's external test releases. [17]

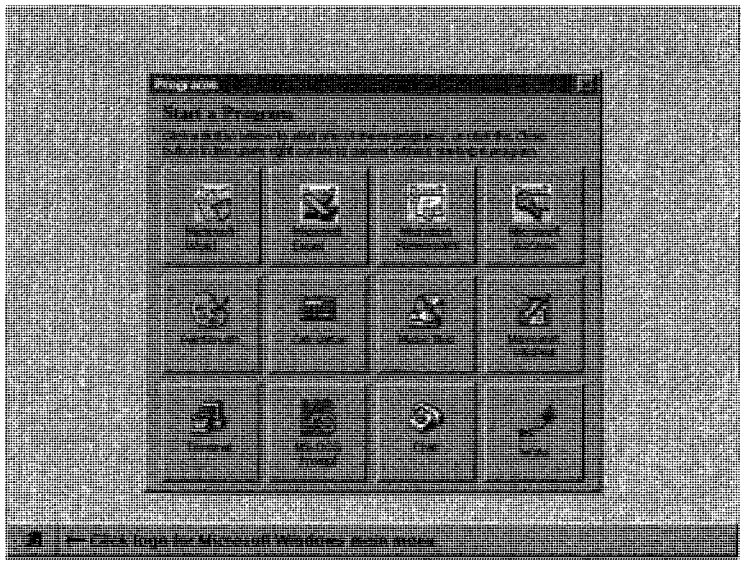


Figure 5-25.
The prototype for the novice shell.

With the final design, you end up with a personal desktop that looks a lot like the older default desktop. The changes to achieve the final default desktop guide the novice into being able to use the system quickly.

17. As of the Beta-1 release, some form of graphical buttons for augmenting the Start menu was still under consideration.

The Taskbar

A number of issues shaped the final design for the taskbar. The main issue was the behavior of minimized windows. The original design, shown in Figure 5-26, had windows shrinking and parking themselves on top of the default taskbar area—although it was still possible to move a minimized window to a different location on the desktop. Then the taskbar buttons became directly related to minimized windows. The final design, shown in Figure 5-27, provides for the creation of a button in the taskbar that corresponds to any window. (Ingeniously, the shell gradually shrinks the buttons as you add more and more of them to the taskbar—the change is almost imperceptible.)

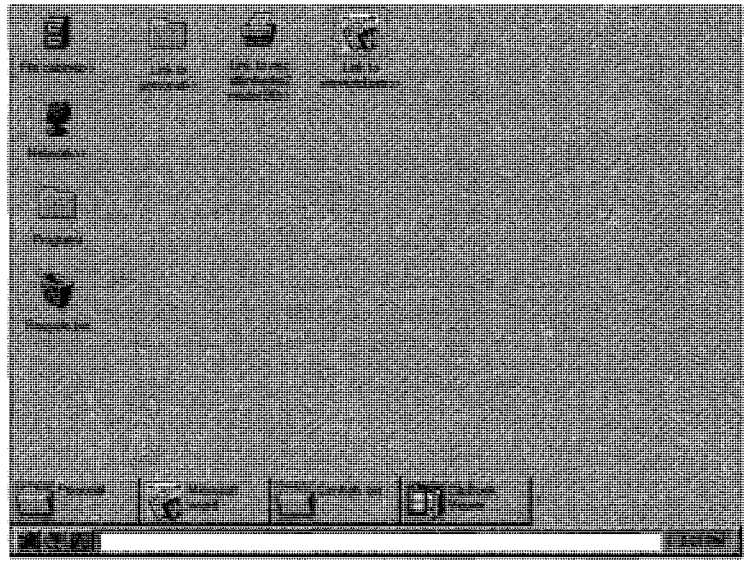


Figure 5-26.
Minimized windows on top of the taskbar area in the early shell.

This final design addresses the user's problems: losing minimized windows and having trouble differentiating among minimized windows, executing applications that simply have very small main windows, and other desktop objects. The user can always go to the taskbar to find an application that is running.

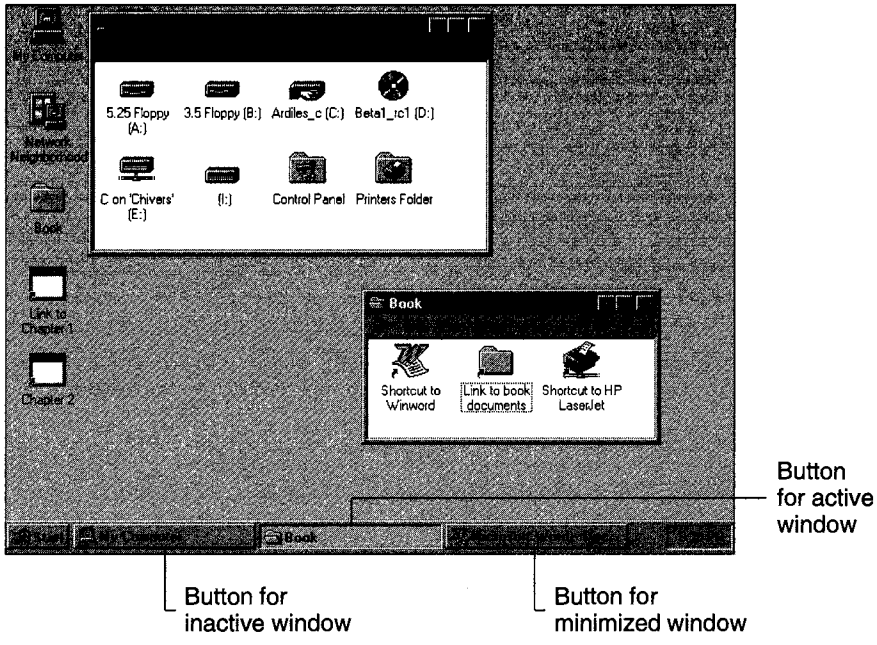


Figure 5-27.
Buttons on the taskbar correspond to all open windows in the final version of the shell.

Folders and Browsing

As you can see in Figures 5-24 and 5-26, the old desktop design in the M4 and M5 releases incorporated a File Cabinet icon intended to be the point of access to local file storage. Not surprisingly, experienced Windows 3.1 users assumed that this was the familiar File Manager application. It wasn't. Under Windows 95, it's the shell that allows you to open folders on the desktop, and the folders can contain any kind of object—not just files. The Windows 3.1 notion of a separate application—the File Manager—that you must run in order to inspect files doesn't really exist in Windows 95.

This subtlety proved difficult for many Windows 3.1 users to grasp, so the designers simplified the shell by altering the file cabinet icon so that it looks like the computer icon you see in Figure 5-27, thus breaking the association with the old File Manager.

The default behavior of the shell resulted in folders opening on top of other folders. Quite soon the desktop would get pretty full, as in Figure 5-7 back on page 170. The modified shell behavior in the final release introduces the explicit Explorer program yóu see in Figure 5-28). The default Explorer behavior displays a two-pane window. Moving through the hierarchy causes the contents of the right-hand pane to be replaced with the contents of the next folder window you open. So more often than not, you'll have just one open folder window on the desktop.

When you browse directly using the shell, there's also an option that allows you to choose either to have a new window for each folder or to replace the current window contents with the new folder. The level of desktop clutter is thus controllable.

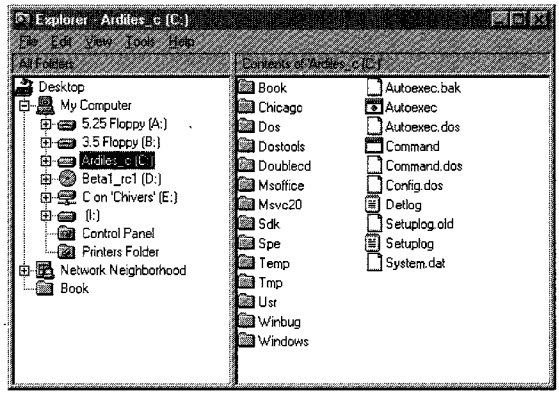


Figure 5-28.
Exploring the system.

Animation

The use of animation in Windows 95 isn't purely frivolous—though it may appear so at first. On the desktop background, the animation effects are there purely for user appeal, it's true. The device was introduced during one of the usability tests, and a lot of people liked it. The popularity of animated screen savers and animated desktop wallpaper seems to lead naturally to animated desktops. (Of course, a whole new third party industry segment will debut, providing replacement desktops for Windows 95.)

The more serious use of animation in Windows 95 is as an indicator of the relationships among objects: a window that shrinks to a minimized state gives the user a pointer that indicates where the application went. A folder that expands into a window showing a list of objects provides a hint that the different objects share something in common. This use of animation is actually quite important to the shell's "explorer" mode. One problem identified in Microsoft's usability tests was the difficulty people had in relating the contents of the left and right panes of a folder view—the tree and the individual folder. Animation helps users relate the contents of the two different panes.

The Transfer Model

Transfer model is the term applied to the user's conceptual view of what's involved in moving information from one place to another. If you know Windows, you'll usually think of information transfers as the Cut/Copy and Paste options found on an application's Edit menu. It's rare to find a document-oriented application for Windows that doesn't support cut and paste operations. Over successive Windows releases, system support for cut and paste operations has been improved both for Windows applications (with the Clipbook introduced with Windows for Workgroups) and for MS-DOS applications in the Windows environment.

Unfortunately, many novice Windows users have difficulty grasping the cut and paste metaphor. A strange hidden application called "Clipboard" is involved, and the user must understand the notion of different data formats to use cut and paste proficiently. With OLE-enabled applications starting to appear, the user's reliance on cut and paste ought to shrink, but there will still be a need to support cut and paste operations for a long time to come.

Microsoft's designers wrestled with introducing a different transfer metaphor, one involving the verbs *move, copy, link,* and *put here.* As you can probably guess, the *move, copy,* and *put here* operations would have effects similar to those of Cut/Copy and Paste, whereas the *link* operation would exploit the new OLE-based ability to support dynamic connections between objects. In fact, OLE uses the new *link* term together with the older cut and paste terms. During the July 1993 design review, these ideas sparked some of the most heated discussions.

Ultimately, the shell designers came to view the problem of redesigning the transfer model as insoluble. Some believed that the new metaphor was conceptually easier for users to deal with, but they also

acknowledged the investment to date in code, documentation, and training that existed for the cut and paste school of thought. The September 1993 internal design review resolved to let Bill Gates decide, with most people leaning toward retaining the cut and paste model.

Other Changes

The most notable change in the shell was the elimination of the "Wastebasket"/"Recycle Bin" feature present in the early test releases. For a number of reasons, this feature was, unfortunately, dropped. Perhaps next time.[18]

The New Appearance

We've already looked at the design concepts that underlie the new look of Windows 95, and you've seen many of the individual elements in the examples. The new look has four main components:

- A more thoroughgoing use of three-dimensional effects. Windows 3.1 does include some 3-D effects on buttons, but Windows 95 uses the 3-D look extensively.
- New system colors and fonts.
- New controls. Windows 95 features several new controls, and these are all available to application programs as well.
- New system dialog boxes. Several of the common dialogs, such as File Open, have been revised.

We're going to take a brief look at all of these items, concentrating on their use in the system. As it did for earlier versions of Windows, Microsoft will publish an *Application Design Guide* book that describes more precisely when, where, and how to use the new visual elements in applications. Many of the new guidelines are manifest in the system itself, and you can find lots of examples in the system of dialogs that have been simplified and generally cleaned up.

Screen Appearance

From the screen shots all through this chapter, you can see that many elements of the Windows 95 interface adopt a 3-D appearance. In Windows

18. Stop press: it's back in, together with a comprehensive Undo feature for all shell operations.

3.1, use of the 3-D effect was limited: most buttons got the treatment, but that was about all. In Windows 95, the 3-D effect is used just about everywhere: for menus, buttons, dialog fields, and more. Of course, this is a 3-D *effect,* not a magical new screen display technology. The main contributor to the effect is the use of different colors around the edges of a screen element.[19]

Figure 5-29 shows how Windows 95 uses outer and inner border color pairs—light gray with black, and white with dark gray—to produce 3-D effects in keeping with the idea of a consistent light source. When a button is not pressed, the top and left edges of its outer and inner borders are in lighter colors than the bottom and right edges of its outer and inner borders. When a button is pressed—as depicted by the outer and inner borders shown in Figure 5-29 at right, by the button shown in Figure 5-30 on the next page, and by the top button shown in Figure 5-31—the top and left edges of its outer and inner borders are in darker colors than the bottom and right edges of its outer and inner borders, and the color pairing of the outer border becomes the color pairing of the inner border and vice versa.

The system augments the basic effects by sometimes reversing the color pairs—pairing black with white, and dark gray with light gray. The outer and inner borders of the pressed button shown in Figure 5-30 are composed of such reversed color pairs. Or the system might pair black

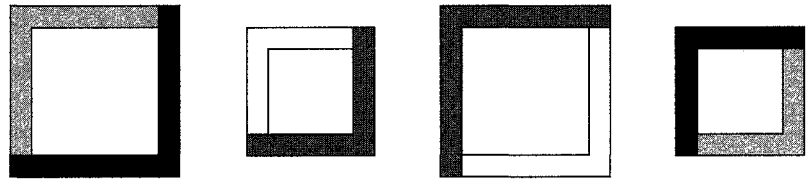
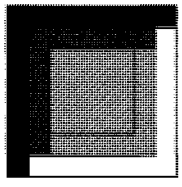


Figure 5-29.
Using the outer and inner borders to create unpressed and pressed buttons in the default color pairs.

19. Exactly why the human eye accepts this simple device as three dimensional is way, way beyond the scope of this book.

with dark gray, and light gray with white. In all, the four colors in three different pairings, combining to show both pressed and unpressed buttons, produce six variations.

Figure 5-31 shows an example of a pressed button as it appears on the screen in the company of unpressed buttons. The user can change the default gray color of the button and the default shading color. If the user changes the default colors, the system supplies the colors it needs to complete the 3-D effect.



Button pressed:
Sunken outer (top and left edges in darker color)
Sunken inner (top and left edges in darker color)
Colors of outer and inner borders exchanged
Color pairings reversed

Figure 5-30.
Using reversed color pairs and creating a pressed button effect.

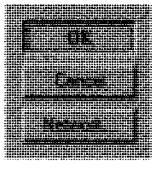


Figure 5-31.
A pressed button.

The other major contributors to the new screen appearance are the different system color scheme and the new treatment of system fonts. Everything is more subtle: gray is often chosen over the stark black and white of Windows 3.1, and fonts are no longer bold.[20] The menus shown in Figure 5-32 exhibit the way in which a Windows 3.1 application automatically inherits these system improvements when it runs under Windows 95. And the new color scheme all actually works on gray scale displays—you don't have to have a 256-color SVGA adapter to realize the benefits of the new look.

20. Microsoft's early test releases of Windows 95 used Arial 8-point regular for the system font.

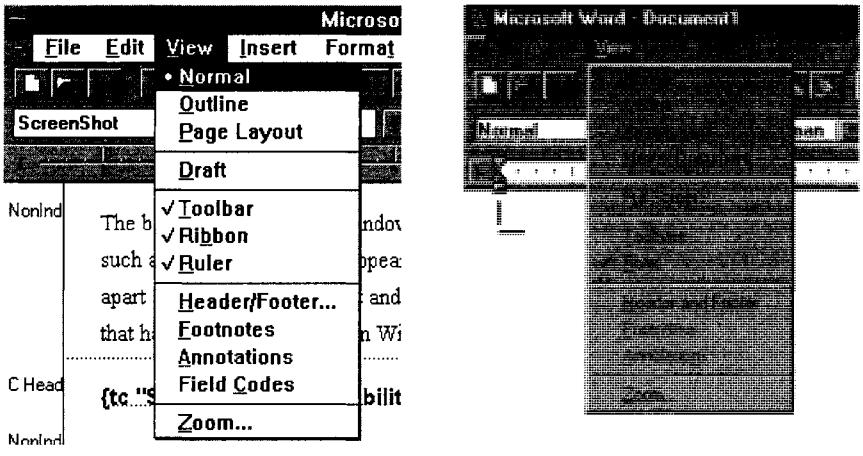


Figure 5-32.
Changes in system fonts between Windows 3.1 and Windows 95.

Visual Elements

The basic elements of the Windows 95 screen are those you're already familiar with from using Windows and applications for Windows. Some of them, such as the tool bar control, appear as standard Windows components for the first time. But you won't find, apart from the property sheet and the new controls it uses, any elements that haven't appeared before, either in Windows or in popular applications for Windows.

Scalability

As part of the overall revision of the Windows screen appearance, the Windows 95 designers did pay a lot of attention to the issue of how to scale the Windows interface. As very high resolution screen displays and adapters have come down in price, their use has grown. Unfortunately, Windows 3.1 doesn't handle this hardware particularly well. Your work might occasionally demand that you use 1280 by 1024 pixel resolution on a 14-inch monitor, for example—at which point, in Windows 3.1, the system font becomes so tiny as to be unreadable and grabbing a window border with the mouse becomes an exercise in patience and dexterity. Similarly, running Windows 3.1 on a very large display tends to result in unnecessarily large amounts of screen real estate devoted to scroll bars and the like. And, of course, the issue of personal preference can't be ignored.

Included in Windows 95 is a control panel for window metrics. You can change the size of every element of a window—even to the extent of making the window's appearance a little ridiculous, as in Figure 5-33.

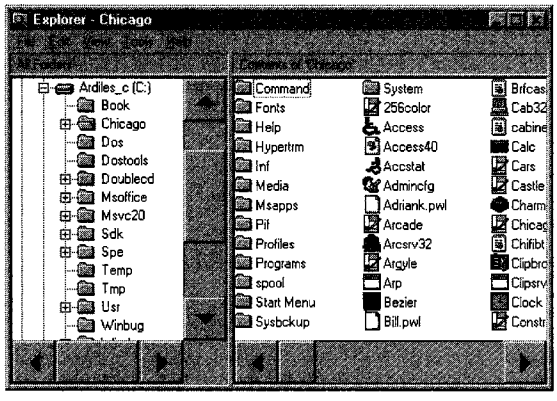


Figure 5-33.
New window metrics in place.

The user can make these changes dynamically: there's no need to restart Windows to have them take effect. One issue application developers have to deal with is the possibility that such changes will occur while an application runs. This problem is similar to that of the user's resizing the system taskbar or to that of dealing with hardware that allows the user to rotate the monitor between portrait and landscape orientation. The video device drivers in Windows 95 also allow screen resolution changes on the fly.

Menus

In addition to the refinements to their colors and fonts, menus have changed in a few subtle ways and a couple of obvious ways. There is also one new menu type: the *popup menu.* The user accesses a popup menu by using the right mouse button (or, more correctly, mouse button two) as he or she selects an object. The popup menu appears next to the object, and the design guidelines recommend that the menu be context sensitive so that it can change according to the current state of the object. Figure 5-34 illustrates the popup menu for a printer that is in the midst of a print operation.

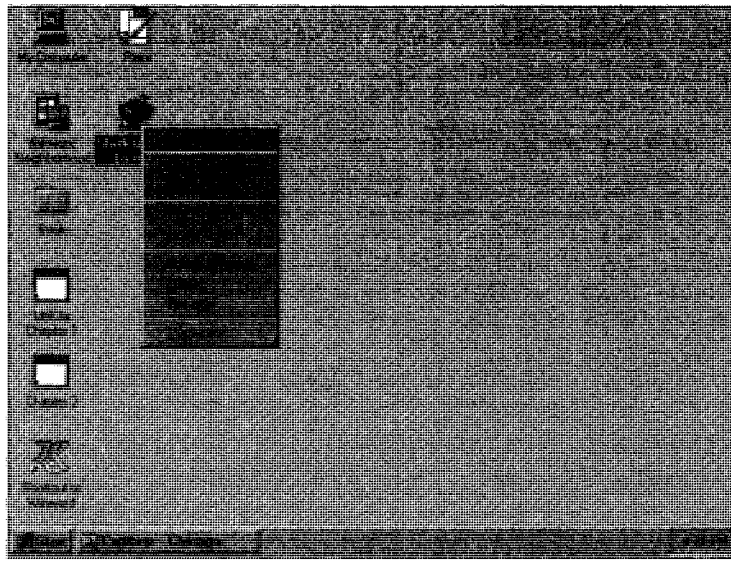


Figure 5-34.
A popup menu brought up by a press of the right mouse button (mouse button two).

The *window menu* is the new name for what you used to call the system menu. The design guidelines add a standard "View" menu that affects the displayed view in the window. Figure 5-35 on the next page shows an example in which the status bar and the tool bar have been turned on using the View menu options.

Of the more subtle changes to menus, the most noticeable is their behavior once you have a menu displayed on screen. Simply moving the mouse along the menu bar will cause other menus to drop down from the menu bar or cascaded menus to unfold from within the current menu. You don't need to click or hold down the mouse button after the first click. This behavior contrasts with that of Windows 3.1, where access to any other menu required at least one more mouse click.[21]

21. Sometimes called a "hot mouse," this behavior has been incorporated into other graphical systems. (It was considered for OS/2 back in 1987 but never implemented.) Most implementations of a hot mouse don't even require the first mouse click—simply passing the mouse cursor over the menu bar makes the menu appear. Some people find this behavior irritating, and others love it.

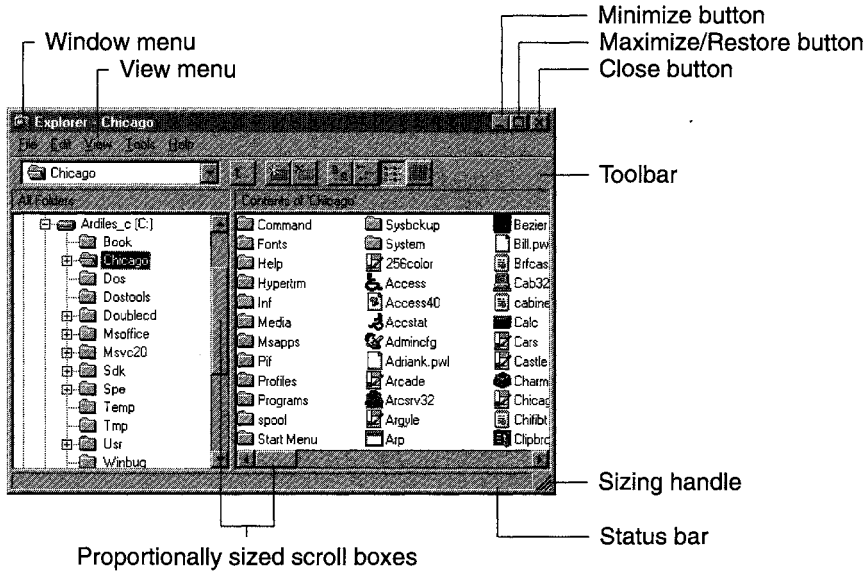


Figure 5-35.
The window display with Tool Bar and Status Bar options.

Window Buttons

The Minimize, Maximize, and Restore buttons located on the upper right of a window's title bar have also changed. The icons depicting the three operations are different. See Figure 5-35 for an example. And a third icon has been added. Clicking this button is the same as doing a Close operation on the Window menu.[22]

Icons

The visual designers have applied the same principles to icon design that they have applied to the rest of the system. The apparent light source for an icon is now the same as for all other controls, and the subtler shading and outlining techniques are used for icons too.

Applications now have to provide two icons: a 32 by 32 pixel icon and a new 16 by 16 pixel size. Windows 95 uses the larger icon to represent the application itself—for desktop shortcuts, for example. The smaller icon appears as a visual aid that can be embedded within a document icon, within a folder's small-icon view (see Figure 5-35),

22. Personally, I disagree with the design decision to place the Close button where Maximize used to be. After you've run a few applications that start with a nonmaximized window, you'll see what I mean.

and within a window's title bar. If the application doesn't provide the smaller icon explicitly, the system will try to create one by scaling down the application icon. Depending on the complexity of the original icon, this may or may not result in a recognizable image.

Proportional Scroll Box and Sizing Handle

To see more or different information in a window, you can do one of two things: scroll the window or resize it so that it has a larger client area. The information Windows 95 displays to help you do this includes a proportionally sized scroll box within the standard scroll bar control and a new sizing handle in the bottom right corner of the window. You can see an example of each of these in Figure 5-35. The position of the box within the scroll bar still provides an indication of your current position in the document. The size of the scroll box shows you how much of the total document is shown in the window. A scroll box that fills the entire scroll bar would tell you that you were looking at the whole document.

The sizing handle is simply a visual cue. Window sizing behavior is the same under Windows 95 as it was under Windows 3.1. If there's no sizing handle, the window is a fixed size.

New Controls

The new Windows 95 controls are available only to 32-bit Windows applications. A 16-bit application can't call the common control DLL that implements the new controls. Many of the new controls are simply standardized system implementations of elements you've seen before in applications for Windows.

Tool Bar Control

With the *tool bar* control, Windows 95 implements perhaps the most popular visual device seen in applications for Windows 3.1. Somewhat as in the garish early days of desktop publishing, applications, including Microsoft's, have sprouted strips of buttons and edit controls that purport to provide a shortcut to every function in an application. Like them or loathe them, they're here to stay. If the Windows 95 tool bar control becomes the preferred method of deploying this shortcut feature, at least we'll have a degree of consistency among different applications.[23]

23. Microsoft's long-term stated direction is to merge the menu bar with a system tool bar. I hope we'll all have 35-inch monitors and excellent pattern recognition capabilities by then.

The tool bar control assists in the management of the buttons on the control. The edit fields, if any, are separate windows. The programmer can add, delete, move, raise, and lower buttons within a tool bar control. The control also supports a customization feature, allowing the user to add his or her favorite buttons to the tool bar. The system arranges for the tool bar control to be automatically resized when the window size changes. Figure 5-36 shows the details of an example tool bar. Figure 5-35 back on page 204 shows an example of how the Windows 95 shell uses the control.

Figure 5-36.
Example tool bar control.

Button List Box Control

The *button list box* control shares some of the tool bar's properties. It allows the programmer to create a horizontal or vertical row of buttons that display application-specific bitmaps. The button list box control might be used to create the floating palettes of buttons popular in some existing applications.

Status Window Control

The *status window* control implements another very popular Windows application and tools user interface component.[24] Figure 5-37 shows an example of a status bar at the bottom of the folder view window. Microsoft Word for Windows used the status bar concept in a very early revision. The status window control allows the programmer to divide a screen area into multiple windows and display text in each of them. Usually, the status bar appears at the bottom of the window, although early API definitions also allowed it to appear at the top of the window. Typically, the text provides helpful information about the current document—the present cursor position, for example. Another common use of a status window control is for a brief prompt to indicate the likely outcome of choosing the current menu item.

24. The Microsoft Foundation Classes for Visual C++ actually included an implementation of the status window control under Windows 3.1.

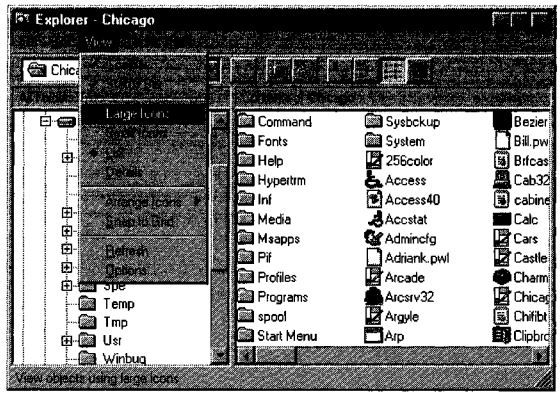


Figure 5-37.
Example status bar.

Column Heading Control

The *column heading* control implements a horizontal window that can include column titles. The programmer positions the column heading window above columns of related information. The user can grab the column dividers within the header window control and drag them to adjust the widths of individual columns. The Windows 95 shell uses the column heading control extensively. Figure 5-38 shows an example of the column heading control's use while the contents of a folder are displayed—the user has substantially increased the default column width for filenames by dragging the column delimiter to the right.

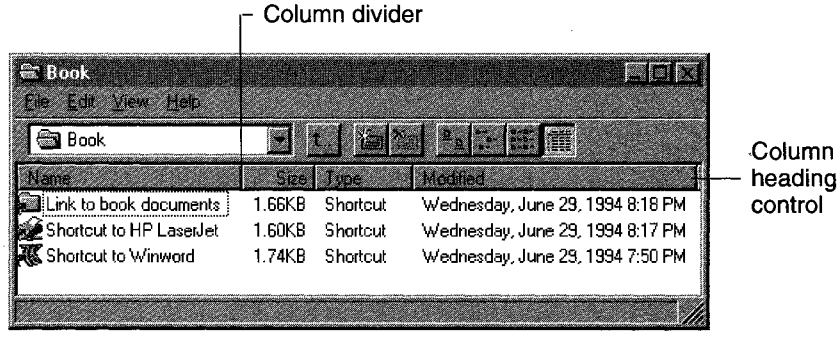


Figure 5-38.
Example column heading control and status window control.

Progress Indicator Control

The *progress indicator* control (sometimes called simply a *progress bar*) standardizes a visual device already used in many applications. It provides the user with an indication of how far a lengthy process is from completion. The application programmer can set the range of the control and the rate of the advance of the current position indicator. If a label for the control is present, it will either show the percentage of the process that is complete or otherwise indicate the current position. Figure 5-39 shows a progress indicator control. You can see an example of its use as Windows 95 scans the disk when you open a new folder.

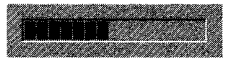


Figure 5-39.
A progress indicator control (progress bar).

Slider Control

The *slider* control is now the preferred control for setting values within a continuous range (as opposed to a series of discrete values). Many applications have used scroll bars for this purpose, but that use was a little misleading since there is no information to scroll through.

The programmer can set the minimum and maximum positions for the control, the tick marks, and the position of the slider. Figure 5-40 illustrates the basic design of the slider control.

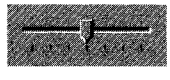


Figure 5-40.
A slider control.

Spin Box Control

The *spin box* control (Figure 5-41) implements a common input device often called a *spin button* or a *spin control.* Clicking on the arrows in the control will alter the value displayed in the associated edit field. As the designers originally defined it, the new control was termed an *up-down* control, and the application programmer had to associate the control with a particular edit control (its "buddy window"). Later discussion seemed to indicate that this division of controls wouldn't come about and that the edit control and up-down controls would be combined into the single spin box control.

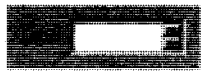


Figure 5-41.
A spin box control.

Rich Text Control

The *rich text* control implements an oft-requested feature: an edit control that allows for the input of multiple lines of text with word wrap and other formatting features.[25]

Tab Control

The *tab* control implements a device that allows the user to navigate among logical "pages" of information. Figure 5-42 shows an example tab control for three pages of information. The most common use for a tab control is within the property sheet control we saw in Figure 5-21 back on page 186. The tab control is meant to suggest to the user a peer relationship among the different pages. If the information is really hierarchical, the dialog organization should reflect that.

Figure 5-42.
An example tab control.

Property Sheet Control

The *property sheet* control implements the mechanism the shell uses to display object properties. Providing the property sheet as a basic control within the system makes it readily available for applications to use. Figure 5-43 on the next page shows a page of the property sheet for an MS-DOS virtual machine control. You can think of each page in the property sheet as if it were a separate dialog box. The buttons at the bottom of the page are global—they relate to the property sheet as a whole, not to a specific page. Every property sheet includes an Apply Now button. Clicking on the Apply Now button will alter the properties to match their new settings but will not dismiss the property sheet (as would happen if you clicked on the OK button). The absence of a strict hierarchy is the major difference between a property sheet and

25. This innovation single-handedly reduces much of the implementation of the Wordpad accessory to the creation and management of a solitary rich text control.

a cascading series of dialog boxes. In a property sheet, you can flip back and forth between pages and leave the property sheet from within any page.

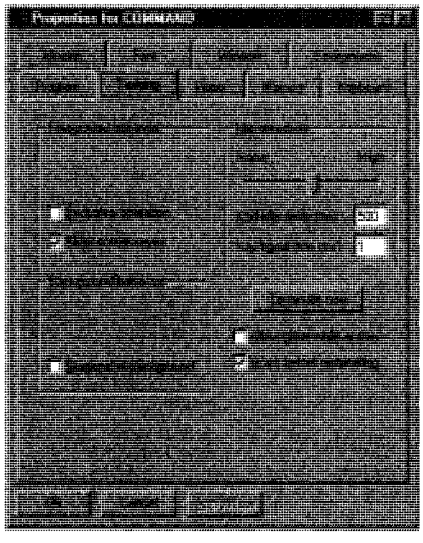


Figure 5-43.
Example property sheet for the MS-DOS virtual machine control, open to the Tasking properties page.

List View and Tree View Controls

The *list view* and *tree view* controls provide the ability to display a collection of items to the user. The shell uses these controls when it displays folders. Figure 5-28 back on page 196 shows examples of both a tree view control and a list view control.

The tree view control provides hierarchical information about items and allows the programmer to expand or collapse parts of the tree. The list view control supports a single-level list of various types: large and small icons and a details view.

New Dialog Boxes

When Microsoft introduced the notion of common dialog boxes for standard operations such as File Open, their actual implementation required the application vendor to ship the DLL that supported the functions. In fact, every Windows application you've installed in recent years

probably came with a copy of the COMMDLG.DLL file. Using the common dialogs meant consistency for the user and less effort for the application developer.[26] These common dialogs gradually became a part of the Windows product. Windows 95 introduces some improvements and some new dialogs.[27]

A few of the common dialogs haven't changed beyond adopting the Windows 95 visual style: the Find and Replace dialog and the Fonts dialog are essentially the same as in Windows 3.1. At least initially, Microsoft planned to make only minor revisions to the Print and Print Setup dialogs. At Microsoft's early user interface design review meetings, however, the audience greeted this plan with something less than tacit agreement. The Windows 95 product release may well include larger scale changes to the print dialogs.

Windows 95 does revise the file management and color dialogs, adds a page setup dialog, and includes all of the OLE dialogs as standard components. Naturally, all of these dialogs exhibit the new visual style, and Microsoft's application design guidelines encourage developers to always use the common dialogs. The Windows 95 common dialogs also use the standard controls (including the new ones we've looked at). Earlier versions of the common dialogs were often built separately instead of making use of the standard controls, and they included some subtle incompatibilities as a result.

File Open Dialog

You'd think that the amount of time and brainpower that have been applied to the apparently simple task of opening a file would long ago have produced the ultimate File Open dialog. Not so. The Windows 95 File Open dialog adds a number of new features to the state of the art:

- The dialog looks very much like a shell folder window, displaying a tree view and a small icon list view of the files and directories.
- You can browse the network directly. You no longer need to understand the concept of network drives to cruise for a file.

26. For a long time, one of Bill Gates's better known complaints was "Why on earth does everyone have to write file open code?" He would usually put it a little more strongly than that.

27. Early examples of most of these dialogs are shown in this section. Some, such as the OLE dialogs, weren't available in time to be included here.

- The dialog includes a document preview window that provides an indication of the file's contents.[28]
- Links and long filenames are understood and handled correctly.
- The dialog provides direct access to an object's popup menus.

Figure 5-44 shows the design for the Windows 95 File Open dialog, which was presented in the first design review meeting. You can see the tree and list views of the folders and documents, the long filenames, and the document preview window.[29]

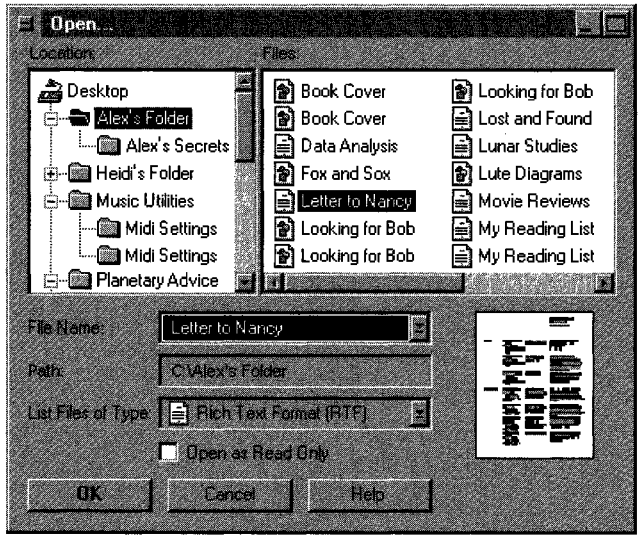


Figure 5-44.
One design for the new File Open dialog box.

28. The intention is to provide a preview window for a very wide range of file types. This goal implies a large number of specialized file viewers and a lot of work—not all of which might get done for the Windows 95 release. One easy file type to display is an OLE compound file, in which the dialog can use the embedded thumbnails directly.

29. The first test release of Windows 95, in August 1993, did not include this dialog.

Page Setup Dialog

Page Setup is a function you see in many applications for Windows. It's not used as frequently as a simple file open operation, but in Windows 95, it makes the cut and becomes one of the common dialogs. Figure 5-45 shows the original design for this dialog. It includes paper orientation and margin setting features, as well as paper handling facilities that used to be part of the Printer Setup dialog.

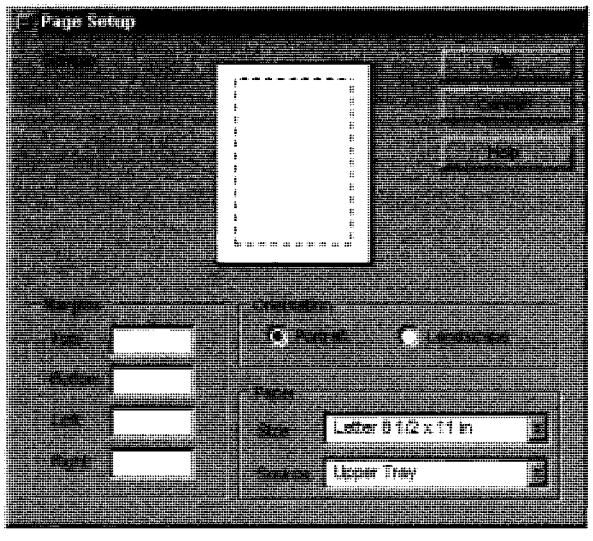


Figure 5-45.
The new Page Setup dialog box.

Long Filenames

In Chapter Seven, we'll look in detail at the new filesystem for Windows 95. The filesystem's biggest impact on the user interface is its support for long filenames. It took a lot of development work to get the shell and other visual elements to fully support this new capability. And if someone chooses to call a file My letter to Aunt Winnie about the dahlias, displaying the name and allowing it to be easily edited becomes a nontrivial task. One new feature of the shell allows document renaming in situ. Figure 5-46 on the next page illustrates the creation of the new filename.

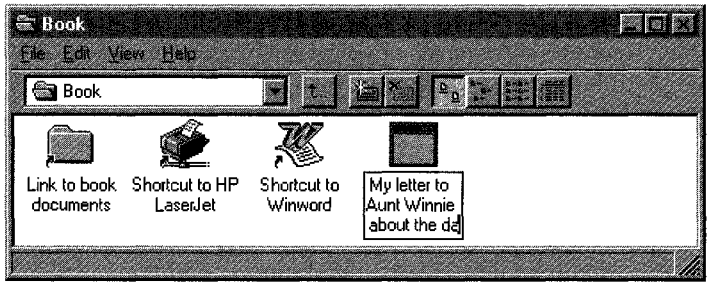


Figure 5-46.
Long filename creation.

Windows 95 and any application written for it will handle the long name quite happily. This is not the case for Windows 3.1 and MS-DOS applications, and Figure 5-47 illustrates how the long filename will appear in Windows 95. The system creates a short name (using the old 8.3 naming convention) that references the same file. If you know the alternative name, you can get at the file. The Windows 95 implementation of COMMAND.COM helps out by listing both the short name and the long name. Figure 5-48 shows the short version of the long filename as it will appear in an earlier Windows application running under Windows 95.

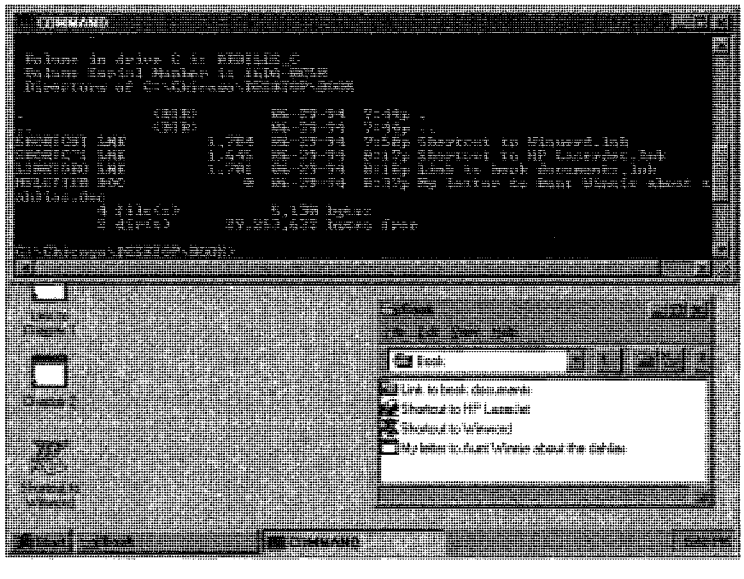


Figure 5-47.
COMMAND.COM in Windows 95 provides a directory listing that shows both the 8.3 version and the long version of a filename.

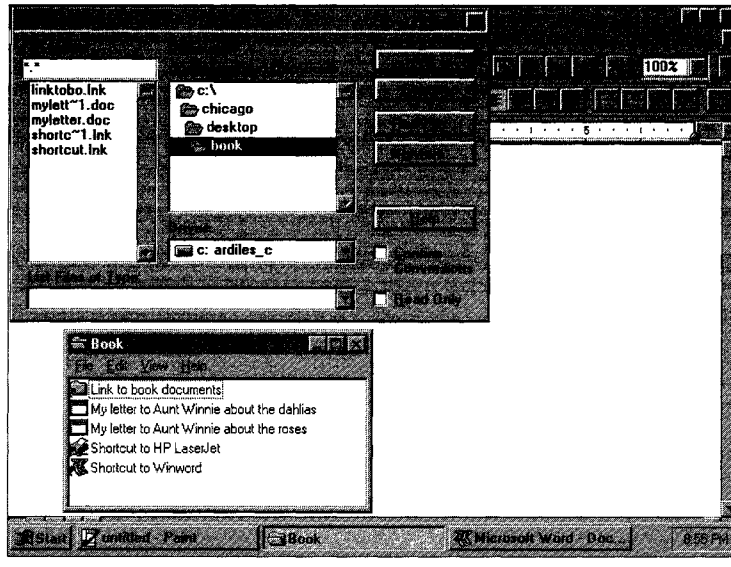


Figure 5-48.
The directory listing in an earlier Windows application running under Windows 95 shows a shortened version of the long filename.

Obviously, with every user's initial mixture of old and new Windows applications, there are going to be some user interface difficulties. This is an unavoidable price that has to be paid if we are (finally) to get the extra functionality of long filenames.

Windows 95 Support for MS-DOS Applications

As one well-known advertising slogan put it, "He's back," or in this case, they're still here. Around the world, beloved MS-DOS applications continue to take up a lot of disk space and CPU time. Acknowledging the obvious, Windows 95 includes some significant improvements to Windows support for MS-DOS sessions—notably:

- COMMAND.COM supports long filenames (as shown in Figure 5-47).[30]

30. Windows 95 also includes new INT 21 API calls that allow the use of long filenames in MS-DOS applications. It will be very interesting to see how many developers revise their applications to support these functions.

- The MS-DOS window is sizeable—just as most other application windows are.
- You can choose the font size for the MS-DOS window. Windows adjusts the font size automatically when you resize the window.
- Windows supports cut and paste operations for any rectangular area within the MS-DOS window.
- The MS-DOS session supports a tool bar control that provides quick access to most of the window functions just described.

Figure 5-43 back on page 210 illustrated part of the MS-DOS VM property sheet you can use to control the behavior of the session. All of the many configurable options are there, along with several new ones. In Figure 5-49, an MS-DOS session window shows part of the tool bar control and one use for the automatic font sizing capability. The font has shrunk so small it's unreadable, but if you're interested only in being able to see when a long series of commands have finished executing (during program compilation, for example), it's sufficient. (After all, you've probably watched that same sequence of commands often enough that you could recite it verbatim.)

Figure 5-49.
MS-DOS application support in Windows 95.

Application Guidelines for Windows 95

Given all the revisions to the Windows 95 user interface, it's not immediately obvious to an application designer what the most important aspects of the new interface are. And for users, there are a lot of new features that require exploring and learning. The success of the product alone will tell whether Microsoft has met its goal of providing a solid transition path for existing Windows users. For the application implementer, Windows 95 includes plenty of new technology to exploit: the 32-bit API and Plug and Play support, for example.

Microsoft recognized the potential bewilderment of the application interface designer and early on in the Windows 95 preview process began to provide design guidelines.[31] The guidelines fell into two categories:

- The user interface style guidelines that had appeared in book form for previous versions of Windows were updated continually throughout the Windows 95 project. The guidelines present a detailed series of recommendations on when and how to use various interface elements: dialog boxes vs. property sheets, for example.
- Guidelines were made available for exploiting the Windows 95 interface to the extent that an application can truly showcase the capabilities of the system.

In each case, there's an interesting question of the lines you have to draw between what you, as an application designer, ought to do or could do as opposed to what Microsoft really wants you to do. Using the common File Open dialog that the user is familiar with is something you ought to do. It makes sense from both a consistency and a cost viewpoint, and the user is likely to consider your application a little strange if you don't use it. Adding support for long filenames is probably a good idea. It costs you implementation dollars, but it's a great feature that enhances any application. OLE support is a feature Microsoft definitely wants to see you add to your application. And it does add an impressive set of features. Unfortunately, it's an expensive addition, and

31. Actually a presentation entitled "How to Be a Great App in the Chicago Shell," which remained fairly consistent throughout 1993.

whether OLE is the way the world will use objects isn't entirely clear yet. Enough speculation—let's take a look at what Microsoft recommends to Windows 95 application designers.[32]

Follow the Style Guidelines

It goes almost without saying that presenting a consistent, predictable environment helps enormously in the user's learning and using applications. It's really what Windows is all about. As we noted earlier in this chapter, Microsoft always points out that their recommendations are just recommendations and not rules. However, many of the guidelines are entirely noncontroversial and make the application design process a lot simpler.

Support Long Filenames

Long filenames are probably destined to be the most immediately popular feature of Windows 95. Given that the system provides much of the basic support for this capability, it looks like a great thing to support in your applications.

Support UNC Pathnames

The number of PCs attached to networks continues to grow at an impressive rate, and Windows 95 is inherently a networked system. Both of these points argue for making applications fully network capable. Support for the *Universal Naming Convention* (UNC) style for filenames is built into Windows 95, and the shell depends on it also for network browsing. Microsoft recommends the support of UNC-style names rather than the drive letter convention. For example, a file open of \\DocsMss\Book\Chapter 5 is preferable to G:\Book\Chapter 5. The preferred title bar caption is Chapter 5 On Docs rather than simply the UNC pathname.

Register Document and Data Types, and Support Drag and Drop

The Windows 95 shell can do a lot without any assistance from the application, provided the application makes the correct resources available, usually by adding information to the Windows registry so that the shell can get at it. In particular, the application helps by

32. In July 1994 Microsoft began to disclose the requirement that an application support many of these features in order to qualify to display the Windows logo. Be warned.

- Incorporating and registering icons for document types to allow the shell to display them correctly when the user opens a folder
- Registering data-specific commands to allow the shell to display the commands in popup menus
- Supporting drag and drop print capability

Use Common Dialogs

The intent of the common dialogs is to provide consistency across applications for frequent operations. The user expects to see the same interface when carrying out one of these operations in any application. The Windows 95 common dialogs also add a lot of features, such as network browsing, that are "free" to applications that use them.

Reduce Multiple Instances of an Application

The perennial lost window problem is exacerbated when an application allows the user to start multiple instances of it rather than simply becoming the foreground application and opening successive document windows.

Be Consistent with the Shell

The Windows 95 shell shows off many of the new Windows features: property sheets, the new controls, popup menus, and so forth. The user will spend a lot of time with the new shell and will come to expect applications to have features similar to the shell's. Providing such features for an application will provide consistency for the user.

Revise Online Help

The style for help in Windows 95 is quite different from the Windows 3.1 help style. Revising the help text for an application so that it will conform to the Windows 95 model is a nontrivial task—a task that may take some time to complete. As part of the revision of online help, Microsoft strongly advocates the incorporation of much more context sensitivity—help popups available in dialogs and help on menu items, for example. As far as the overall revision of help systems is concerned, the general philosophy of Windows 95 help is for task orientation and brevity. So don't use a request for fonts help to embark on a discussion of scaling technologies; tell the user how to choose a font.

Support OLE Functionality

The move to objects is on, and Microsoft wants you to view the object-oriented world through the capabilities of OLE. Although OLE is not without its competitors, the support for it from Microsoft's (extremely successful) operating system platforms gives it a definite edge. In particular, Microsoft has based a number of concepts for the Cairo system on work originally done by the OLE group.

Several applications have already incorporated OLE technology, and the resultant functionality is impressive. Right now, adding full OLE support to an application is an extremely complex engineering project. New development tools and methods will no doubt reduce the cost of OLE implementation. If you do use OLE within an application in combination with Windows 95, you'll get these features:

- The OLE compound file as the application data type allows the shell to display the document properties such as the thumbnail view. This compound file format will be the native format for Cairo, so there's another incentive to support OLE now.
- OLE drag and drop will allow users to move and hold documents anywhere in the shell's workspace—the desktop will be the most common place in which to hold them.
- The OLE in-place editing capabilities preview the move to component software and the document-centric interface that Windows 95 promotes.

OLE is leading edge technology. Using it now is expensive but could also give you a competitive edge in the Windows 95 applications market.

Conclusion

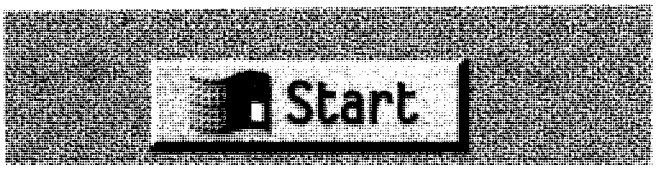
In this chapter, we've taken a lengthy tour through the most visible part of the system. As the Windows 95 visual designers are wont to remind people: details count. Many details of the interface have changed, and several new or improved concepts make their debuts in Windows 95. The biggest change from Windows 3.1 is evident in the shell itself. Given Microsoft's intention to provide Cairo with the same user interface, it will be interesting to see whether the new shell achieves the dual goals of making the system easier for novices to use and providing a natural transition for experienced Windows users.

We haven't looked at some components of the shell in this chapter—the desktop accessories, for example. And you'll have to take your own tour of Windows 95 to see a lot of the more detailed revisions to specific dialog boxes and utility programs. But we did look at all the important new pieces with the exception of the pen interface. Windows 95 includes support for pen computers within the basic system—pen support no longer comes from an add-on module as it did for Windows 3.1.

Now we have to dive a little deeper into the system. In Chapter Six, we'll look at the details of the graphical environment supported by Windows—at how applications harness the graphical environment and how devices are commanded to display it.

Reference

Microsoft. *The Windows Interface—An Application Design Guide.* Redmond, Wash.: Microsoft Press, 1993. This book appeared as part of the Windows Software Development Kit and as a separately published volume. It's the final word on how a Windows 3.1 application should look and contains a lot of useful insight into user interface design. The Windows 95 team produced an updated version of this book under a new name, *User Interface Design Guide,* for the Beta-1 release and planned to update it for the Beta-2 release. Microsoft Press will publish a final version titled *The New Windows Interface.*



CHAPTER SIX

APPLICATIONS AND DEVICES

In Chapters Four and Five, we looked in detail at two of the major Microsoft Windows enhancements that appear in Windows 95: the 32-bit protected mode base operating system and the new user interface exemplified by the shell. The improvements in the base OS help support many collateral enhancement details in the Windows subsystem, and the shell with its new features is but one manifestation of the new capabilities you'll see in Windows 95 applications. To realize these enhancements, applications call on the Windows API, and when the user interacts with an application, a requested service is translated into some device-specific operation, such as the manipulation of visible objects on the display screen or the reading of information from a disk file.

Many different software modules are involved in the translation of user and application actions to specific hardware operations. In this chapter, we'll look at some of the most important components: at the Windows 95 API and its implementation in the Windows User and GDI modules and at a few of the device drivers and subsystems associated with the User and GDI modules. If you need a primer on the basics of how Windows implements its graphical environment, see Chapter Three. Our concentration in this chapter will be very much on the new and different features of Windows 95:

- The Win32 API implementation in Windows 95
- Enhancements in User, the window management subsystem
- Improvements in GDI and the associated graphical device subsystems that control the display and the printer

In later chapters, we'll look at the major product enhancements for local and network filesystem support. This chapter is biased in the direction of what Windows is best known for: its graphical application environment.

The Win32 API

During 1993 and 1994, Microsoft invested enormous amounts of its developer relations time and effort in promoting two specific elements of its operating system products: the Win32 API and OLE. If you left any of the company's systems software presentations with any doubt about what Microsoft wanted you, as an application developer, to develop for, the incessant Win32/OLE chant must have put you to sleep. Naturally, business reasons were at the base of this promotion: if most of the industry's applications are written for your operating system interface, you get to sell the most operating systems. The history of MS-DOS and Windows bears this out. But as the histories of UNIX, OS/2, and indeed early versions of Windows attest, convincing developers to invest resources in a new API is extremely difficult. So Microsoft put everything it could in gear to sell the Win32 API, starting with Windows NT and now with Windows 95.

The Win32 API has one big advantage in its favor: it is by and large compatible with today's most popular API, the Windows 3.1 API. The Win32 API is also extensive. With well over 2000 functions and macros and having undergone a few years of field trials, Win32 offers a wealth of features.

Microsoft's first implementation of Win32 was released in 1992 as the Win32s add-on for Windows 3.1. Recognizing that the rate of adoption for Windows NT would be governed largely by the availability of true 32-bit applications, Microsoft released the Win32s subset to give developers an early opportunity to begin porting their code to the Win32 API. With the release of Windows NT in mid-1993 the first full implementation of Win32 came to market. During the rest of 1993 things got a little more confusing. Later in that year Microsoft began to talk about Win32c—that "c" initially meaning "Chicago" and later spun to "compatible." Eventually the "c" was dropped and Microsoft began to talk simply of different implementations of the Win32 API—each particular to the underlying operating system.

As a practical matter, the Windows 95 implementation of Win32 will probably come to be seen as the "standard" implementation—if

only because of the size of the Windows 95 market. As a numerical matter, the Win32 APIs implemented in Windows 95 account for 95 percent of the total defined Win32 interface. The APIs missing in the Windows 95 implementation are specific to capabilities that Windows NT has and Windows 95 does not—the rigorous security features in Windows NT, for example. But the Windows 95 implementation introduces features that the Windows NT version 3.1 implementation doesn't include—for example, the new device-independent color capabilities. No, this doesn't mean another round of subset and superset confusion. Microsoft plans to promptly update Windows NT so that it will retain its position as the provider of the full Win32 API.[1]

In addition to the API compatibility issue is the issue of binary compatibility: the different operating system products must be able to load and run the various flavors of Win32 application. Both Windows 95 and Windows NT will load Intel format Win32 binaries and run them as full 32-bit applications. Windows 95 will never have a non-Intel processor implementation of Win32. Only Windows NT will run applications compiled for other processors.

What's a developer to do? If you believe in the continued success of Windows, you have to develop for that platform. With Windows 95 we'll see the arrival of full 32-bit support for a mainstream operating system, so if you're starting from scratch, Win32 is the way to go. Since the new features of Windows 95 are available only to Win32 applications, porting your 16-bit Windows code to the Win32 API is an obvious first step. Fortunately, the tools Microsoft provides to assist in the porting task make it less than onerous. Beyond that, the OLE mountain looms—although improved versions of Microsoft's Visual C++ (among other language products) are making that assault a little easier.

All of this begs the question of whether Windows really is the right platform to develop for. It's hard to argue against the current commercial success of Windows, and all of the pieces are falling into place to ensure a continuation of that success. No doubt the debate will continue in many quarters, however. In the meantime let's take a look at what Microsoft is trying to achieve with the Win32 API.

1. Some of the new color facilities will appear in the next release of Windows NT—the so called "Daytona" product. Others will appear as add-on libraries when Windows 95 ships.

Goals for Win32

Microsoft's overriding desire is to concentrate both its own efforts and those of other developers on a single, long-lived API. As candidates for the base API, the existing APIs for both MS-DOS and Windows 3.1 fell short in several ways: they weren't portable, they weren't 32-bit, and they were functionally deficient. At one time the OS/2 API was supposed to be "the API for the future," but for many reasons that prediction didn't work out too well.

A single API does accelerate the market. More people write more software, resulting in more users finding satisfactory solutions to buy. This is one of the reasons MS-DOS was so successful. The PC world had gotten very complex since the first release of MS-DOS, though, and Microsoft decided it was time to try to re-introduce a little more order. Enter Win32—an API aimed at meeting the following goals:

- Broad support. Meeting this goal entails developing plenty of developer momentum and getting lots of applications released in as short a time frame as possible. The best way to do this is to make Win32 as closely compatible with Win16 as possible.[2] Porting applications from Win16 to Win32 will thus be simplified, and momentum will quickly build.
- Portability. Windows NT was designed as a portable operating system—specifically to allow it to run on RISC processors. The debate over whether and when the Intel processor architecture will finally be outperformed by RISC technology continues. Irrespective of the outcome in the hardware battle, Microsoft aims to establish Win32 as the preferred API.
- Room for growth. As PC technology continues to improve, the operating system must be able to offer access to the improvements. Whether the technology be high-speed video on demand or radio-based networking, Microsoft wants an API that can be extended to support the new technologies without modifications to the existing interfaces.

2. The fact that the OS/2 Presentation Manager API differed so widely from the Windows API (both conceptually and syntactically) was a major factor in the slow adoption of OS/2. The Win32 developers, many of whom were involved in the PM effort, were careful not to make the same mistake twice.

- Scalability. Windows NT supported multiprocessor machines in its first release. There's already news of processors that operate with a native word size of 64 bits. The era of the PDA has begun. Developing software for all of these hardware platforms would be impossible if the software platform were different for each. One API suitable for supersetting and subsetting for different hardware platforms will help a lot.

Components of the Win32 API

Before we examine the details of the Win32 API, it should be worthwhile to look at a few of the statistics and then to group the functions. Bear in mind that the statistics deal with a prerelease of the product some months before its expected release. The absolute numbers will probably change, but the proportions should stay roughly the same.

As of this writing the total number of Win32 APIs, macros, messages, and defined constants is 2246. Of these members, 1350 were included in the Win32s subset and only 114 are not in the Win32 API set supported by Windows 95. Of the 114 members supported only by Windows NT, almost all relate to the security features or the service control and event logging subsystems available under Windows NT. Of the 2246 total, 546 of the interfaces are macros, messages, and predefined constants, so the API total drops to a very manageable 1700 interfaces!

The major components of Windows 95 remain the Kernel, User, and GDI modules that provide the interface to the base OS, window and application management, and the graphics facilities, respectively. Each of these modules supports about 300 APIs.[3] In Windows 95, these APIs are the major extensions to the three basic modules:

- OLE. The OLE APIs, numbering only (!) 66. They are perhaps the most complex and, for Microsoft at least, the most important extension of the core Windows system.
- Controls. The support for the standard user interface elements described in Chapter Five.
- Common dialogs. Dialogs such as "File Open" that are shared by applications.

3. To be precise, in the M5 version it was 346 in Kernel, 262 in User, and 300 in GDI.

- Decompression. File decompression capabilities commonly used during installation.
- DDE. The Dynamic Data Exchange facility. DDE was Windows' first popular application information interchange capability. Over the course of time OLE is expected to replace the use of DDE.
- RPC. The support for remote procedure calls relied on for distributed application development.
- Sockets. The so called "WinSock" interface. Sockets has grown in importance for Windows networking. Originally developed simply for TCP/IP network support, Sockets is now seen as the best way to develop non-RPC network applications for Windows.
- Networking. Network-specific APIs outside the RPC and socket interfaces. Of course, many of the Kernel APIs ultimately find their way to the network subsystem for file input/output and other operations.
- Communications. A set of APIs designed to support reliable wide area communications applications such as electronic mail and remote network access.
- Shell. A set of APIs supported by the shell itself that enables the extension of the shell's capabilities through installable libraries.
- Multimedia. Extensions to the core system for audio and video management. The multimedia extensions number close to 200 APIs—interestingly the largest single set of extensions.
- Pen. Extensions to the core system that support the specific needs of pen-based applications.

As mentioned earlier, that won't be the end of the Win32 API story. Already, Microsoft has begun to describe its plans to implement the OpenGL 3-D graphics library for Windows NT—a component that will add another 300 or so APIs to Win32. But for the purposes of this chapter's discussion we'll concentrate on the core components that we haven't yet examined: User and GDI.

The Win32 API on Windows 95

Developing a Win32 application for both Windows 95 and Windows NT requires that you recognize two basic kinds of issues: those inherent in porting existing 16-bit code to the 32-bit interface, and the Win32 APIs that aren't supported on Windows 95. In addition you can observe some general programming guidelines that help prepare an application for future improvements—after all, someday you may actually have to worry about 64-bit interfaces.

Porting to the Win32 API

You'll find extensive documentation describing the details of the 16- to 32-bit porting process in the Windows SDK products, so there's little value in a regurgitation of all of it here. A few of the more important aspects are worth reviewing, however: notably,

- The mechanics of the porting process
- API syntax changes
- Memory management
- Version checking

Note too that if you're tempted to try to mix 16-bit and 32-bit code (using the Microsoft thunk compiler tools) to help speed up the porting process, you'll end up with an executable program that will run only on Windows 95 and that won't even load on Windows NT. You'll also create the potential for many bugs because of the different sizes of integers (and thus of many Windows data types). Microsoft's recommendation is simply don't mix 16-bit and 32-bit code segments. If you have to mix them, make sure that the 16-bit code is carefully isolated and plan to replace it as soon as you can.

Porting Tools

If you're starting with a 16-bit Windows application, there's some mechanical help at your disposal. Included in the Windows SDK is a source code analyzer called PORTTOOL.EXE that will examine each and every Windows interface and suggest changes you may need to make. This porting tool isn't foolproof, but it's a good way to start the process. Another mechanical aid is to define the *STRICT* constant

when you compile your code. Then the strictest level of type checking will be applied to Windows functions. Your fixing the ensuing stream of warning messages can often remove subtle bugs before they have a chance to bite.

The WINDOWSX.H header file included in the SDK also contains many macros that cloak API calls in a single portable interface. If you have to maintain both 16-bit and 32-bit versions of an application, that's some help.

API Changes

As successive versions of Windows have appeared, more and more parameterized types have appeared in the declarations of Windows interfaces. Most programmers are familiar with declaring device context handles as *HDC*, for example, but the "before" and "after" declarations of the main window procedure shown in Figure 6-1 illustrate just how pervasive the technique has become with Win32. Admittedly, the person who wrote the "before" declaration must not have touched the code in a very long time, but the new types in the up-to-date version affect every part of the declaration.

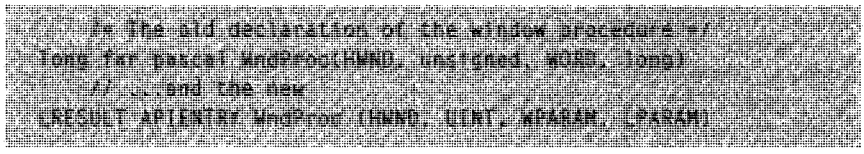
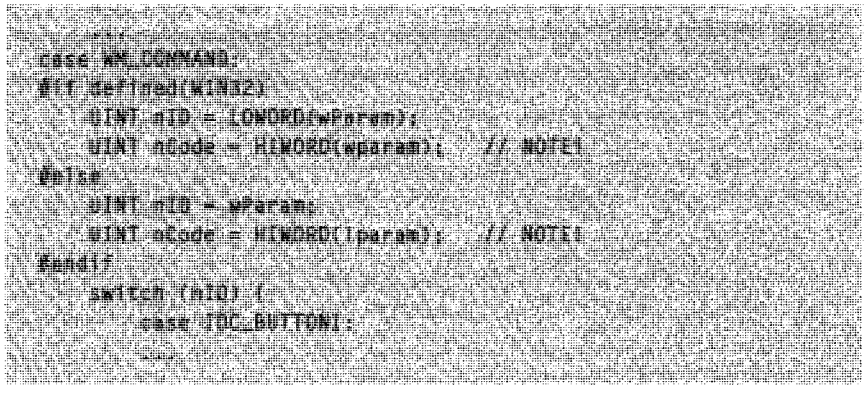


Figure 6-1.
Using predefined types in Win32.

Modifying the code this way assists in compiler type checking and also masks the actual word size of the underlying system. Unsigned integers that were 16-bit quantities are now 32-bit values—and can become 64-bit values with no further code modification. This *widening* of many 16-bit values can be seen in a lot of the Win32 APIs. It's really an artifact of the extensive use of C integers: they were 16 bits on Windows 3.1, and they become 32 bits on Win32. But the changes aren't purely syntactic. There are some semantic issues as well.[4] Figure 6-2 illustrates

4. There's also the subtle issue of alignment: structure fields that lined up neatly on 16-bit boundaries may not do the same when integers widen to 32 bits. On the 386 this results in only a slight performance overhead, but on some RISC processors it causes a hardware fault.

one of the porting problems engendered by the Win32 API that can't be fixed simply by careful use of the predefined types.[5] Here the data supplied with the WM_COMMAND message has been packed into the *wParam* and *lParam* parameters differently, necessitating code that differentiates between API versions. This sort of change between Windows 3.1 and Win32 is not uncommon. The porting tool helps you find the occurrences, but even so this is one area in which careful checking is necessary.



```
    ...
case WM_COMMAND:
#if defined(WIN32)
    UINT nID = LOWORD(wParam);
    UINT nCode = HIWORD(wparam);   // NOTE!
#else
    UINT nID = wParam;
    UINT nCode = HIWORD(lparam);   // NOTE!
#endif
    switch (nID) {
        case IDC_BUTTON1:
        ...
```

Figure 6-2.
Message parameter passing in Win32.

You'll also see many Win32 APIs with names similar to those of Windows 3.1 APIs but with an *Ex* suffix. Microsoft has used this convention to signal that it's extending the functionality of an existing Windows 3.1 API in some minor way.[6] The recommendation for porting code to Win32 is to use only the APIs with the *Ex* suffix. You'll find the superseded function marked "deleted" or "obsolete" in the Win32 documentation. Figure 6-3 on the next page shows one example, the GDI function for setting a window origin. The old version has been modified to return the coordinates of the previous window origin differently.

5. *#ifdef*'d code never was the best way to handle this sort of problem. You can write portable code to handle either situation. The *#ifdef* method makes for a better illustration, though.

6. Unfortunately, neither the extent of the extending nor the name signal are entirely consistent. A few of the extended functions incorporate major additional functionality. And some extended functions have *Ext* as the *prefix*, not a suffix, for the old name. The Windows API naming story continues.

```
    // The obsolete version. Coordinates returned
    //  as the result of the function
DWORD SetWindowOrg(HDC, int, int)
    // ...and the Win32 version. Function result now
    //  denotes success or failure, and the previous
    //  origin is stored in the LPPOINT parameter
BOOL SetWindowOrgEx (HDC, int, int, LPPOINT)
```

Figure 6-3.
Similar function changes in Win32.

Most of the extended APIs are GDI functions, and the *Ex* form of the API was actually included in Windows 3.1. The difference is that the older form of the function call is unavailable in Win32. Windows 3.1 actually supported both. The GDI functions also mask one important difference between Windows NT and Windows 95: the difference in their graphics coordinate systems. On Windows NT you identify a point using 32-bit coordinates. Windows 95 retains the older 16-bit coordinate system. For graphics-intensive applications this is an important difference that is syntactically manageable by means of the predefined types (predominantly POINT and SIZE structures). But the associated semantics are a different matter, with no easy solution for developers who would like to exploit the capabilities of the 32-bit coordinate system on both Windows 95 and Windows NT.

Memory Management

We looked at many of the new aspects of Windows 95 memory management in Chapter Four. Apart from the new features, from the application programmer's viewpoint, the Win32 API makes things a whole lot easier. Segments are now a relic, so it's good-bye to far pointers, and any other vestiges of Windows' 16-bit past, such as having to lock and unlock memory objects, can be dispensed with.[7]

The fact that the system is now entirely virtual memory based means that the absolute addresses or contiguous locations of certain segments are no longer the same under Windows 95. The addresses and locations were never published and ought not to have been assumed, and under either Windows 95 or Windows NT, the rules change. You absolutely must use the defined memory management APIs if your code is to work correctly.

7. If you did atrocious things with direct segment arithmetic, it's payback time.

Version Checking

Microsoft chose to handle the Win32 API subset issue on Windows 95 by actually implementing the full set of Win32 APIs and then returning an error if a call is made to an API not supported by Windows 95. This strategy allows a Win32 application to always load under either Windows NT or Windows 95—references to missing DLL entry points don't stand in the way. But if you call an API that exists only in the full Win32 set on Windows NT, you must be prepared to deal with an error return on Windows 95.

Calling the *GetLastError()* API in response to the error return indicating a failure and getting the ERROR_CALL_NOT_IMPLEMENTED error code will tell you that you've called an unsupported API. A *GetVersion()* API enables you to identify the particular version of Windows that you're running on.

In a very few cases, an API that isn't really supported by Windows 95 will run without the return of an error. One example of such an API is the *GetThreadDesktop()* API that under Windows NT will return a handle to the desktop window associated with a particular thread. Windows 95 has only one desktop, so it's always the same handle that gets returned. Since no undesirable side effects of using this API on Windows 95 are possible, it's easier to allow the call to succeed than to insist that the application handle an error return.

Nonportable APIs

Although some of the older Windows APIs have vanished, the presence of their direct descendants in Windows 95 ensures that porting existing 16-bit Windows code will be a manageable chore. The only snag comes from the use of MS-DOS functions within Windows-based applications—by means of the provided *Dos3Call()* API of Windows 3.1 or by means of embedded assembly language code that calls MS-DOS directly. Win32 doesn't support a direct MS-DOS interface, and it never will. Even if translating 32-bit parameters to 16-bit equivalents weren't an issue, the fact that the base operating system in Windows 95 is entirely call based and makes no use of the Intel software interrupt mechanism other than for compatibility when Windows 95 is running older MS-DOS applications means that Win32 applications that issue MS-DOS software interrupts will fail. If you have code that calls MS-DOS directly—for file I/O, for example—you have to replace the call with the appropriate Win32 API.

Win32 on Windows 95

We'll look at some details of the API changes and enhancements a little later, when we take a closer look at the User and GDI modules. First let's see what Windows 95 doesn't implement that Windows NT does implement. Remember, it's all Win32. As the design of Windows 95 progressed, the Win32 specification changed to accommodate new features that would come to market for the first time with Windows 95. Whether the Windows NT API comes to be regarded as a superset of the Windows 95 API, or the Windows 95 a subset of the Windows NT remains to be seen.

Faced with the prospect of turning all of the new ideas and the enhancement requests into specific Win32 APIs, Microsoft had to consider a couple of factors over and above the basic design and implementation challenge. Was the underlying operating system capable of fully supporting a proposed feature? Was the feature appropriate for the intended market? By and large, you can see these criteria reflected in the eventual choice of APIs that would not be fully supported by Windows 95.

Security APIs

The collection of Win32 APIs that deals with system security issues is merely the most visible aspect of the security capabilities embodied in Windows NT. The system implements stringent authentication and privilege checking features that allow it to be used for secure applications: in a network server role or as a C2-compliant desktop system.[8] For the system to be fully secure, you must use the NTFS filesystem with Windows NT—since the FAT filesystem is provably insecure.

The Windows NT internal system architecture is dramatically different from the Windows 95 architecture in order to meet the secure system goal. This difference translates into a need for more system memory and more processor horsepower—more than the average target Windows 95 machine would have. Since the underlying operating system can't fully support them, Windows 95 does not implement the Win32 security APIs. Microsoft's reasoning: why try to provide two products to meet the same need? If you really need the security capabilities, you'll know it—and you'll use Windows NT.

8. Windows NT on its own cannot be C2 certified. The certification process requires a complete system—the hardware, the operating system, and applications—to undergo verification.

Console APIs

The Win32 console APIs provide an environment for applications that require character mode I/O facilities. For applications with simple user interface requirements—a compiler, for example—the console APIs offer an easy way to run using Win32.

Windows 95 supports the console APIs but provides support for only a single console subsystem. Whereas Windows NT allows the management of multiple console sessions by means of the *AllocConsole()* and *FreeConsole()* APIs, Windows 95 supports only a single console session.

32-Bit Coordinate System

There is no *world transform* coordinate transformation capability in Windows 95, and neither the associated *SetWorldTransform()* and *GetWorldTransform()* APIs nor the XFORM data structure is supported in Windows 95.[9] Their absence is tied to the decision to retain a 16-bit coordinate system in GDI. Implementing 32-bit coordinates really requires a full 32-bit GDI, which, partly for memory consumption reasons and partly for timescale reasons, Microsoft chose not to implement for Windows 95.

Unicode APIs

The first release of Windows NT was unusual in that it supported the Unicode character set specification not only for applications but also as its own internal character set representation. Every Unicode character requires 16 bits for storage—which expands the system's memory requirements—and in addition many compatibility considerations are associated with existing character strings: filenames on disk, and 16-bit Windows application resources, to name just two.

Supporting Unicode would have been a big leap of faith for the Windows 95 team to take. They chose not to, so the system retains its ANSI character set roots and doesn't support the Win32 Unicode APIs. However, some new aspects of the Windows 95 system do use Unicode internally: its long filename support in the filesystem and its 32-bit OLE subsystem, for example. And Windows 95 has far more extensive support for international versions of applications than any of the earlier Microsoft operating system products.

9. If you use world transforms, be sure to read up in the Win32 documentation on how the *SetGraphicsMode()* API works under Windows 95.

Server APIs

The Windows NT role as a highly capable network server means that there are groups of Win32 APIs supporting server operations: notably, server-side named pipes and RPC facilities and tape backup APIs. The server-side named pipes allow a server process to create a pipe that multiple client processes can connect to. The RPC facilities you won't find in Windows 95 include the locator and endpoint mapper features.[10] These features relate to the name service facilities provided by the full Win32 API. (Windows NT supports an endpoint mapping service, RPCSS, and a locator service that don't exist on Windows 95.)

Printer Support

Windows 95 doesn't include the entire gamut of print APIs defined for Win32. There is no forms support (all the APIs with *Form* in their names), and the *AddJob()* and *ScheduleJob()* APIs available on Windows NT aren't supported either.

Service Control Manager APIs

Windows NT supports a *service control manager* facility that allows a subsystem, such as a network server, to register itself as a *service*. Once the subsystem is registered, the system itself takes care of starting the service and maintaining information about currently running services. Under Windows NT, the service control manager is actually accessible across the network by means of RPC, so it's possible to manage networkwide services from a single machine.

In an oversimplification, you could say that the Windows NT service control manager is a highly structured form of the capabilities inherent in the startup files you're familiar with, such as AUTOEXEC.BAT and WIN.INI. The general philosophy of the service control manager doesn't really fit a personal system such as Windows 95, so the service control subsystem and the associated Win32 APIs aren't supported.[11]

Event Logging

Associated with the service control manager are the event logging facilities. Under Windows NT, these facilities allow subsystems to record

10. The full Win32 RPC also includes some Unicode and security related APIs. As you'd expect, these aren't supported on Windows 95.

11. Service control APIs are generally recognizable by virtue of the *Service* or *SC* in their names. For once there's some orderly naming going on.

information about interesting occurrences: unexpected errors, configuration changes, and the like. The Windows NT administrator can inspect the event log when trying to diagnose problems or simply to verify the health of the system. Windows 95 doesn't support the Win32 event logging APIs.

Detailed Differences

Within the Win32 API a number of details have been changed or enhanced and that will affect some applications. Later we'll look at some of the brand-new Win32 features and at Microsoft's recommendations for application developers. Here are just a few of the lower level modifications:[12]

- Most application resource limits have been substantially raised: memory, handles, and other resources are all plentiful under Windows 95. There are 32,767 window handles, for example, compared to only 200 in Windows 3.1. Similar improvements have been made for COM and LPT devices, with Windows 95 providing many more logical ports than physical ports. Total available memory rather than individual resources now becomes the limiting factor.
- Windows 95 tags every application resource with the thread identifier of its owner. When an application quits, the system automatically frees all resources that have been allocated to the application. Some Windows 3.1 applications assume the continued allocation of a resource even after an application terminates. Such an assumption is not valid with Win32 applications.
- Windows 95 includes yet more parameter validation. Whereas Windows 3.1 concentrated on validating the parameters supplied to the published APIs, Windows 95 also validates the so called "undocumented" interfaces that have been discussed in various books and journals. If you use undocumented interfaces, beware.

12. Naturally, the detailed information about these changes tends to be spread around in the documentation. One way of pointing yourself in the right direction is to look for the string *#if (WINVER >= 0x400)* in the Windows SDK header files. The Windows developers have used this string to bracket all the new definitions.

- There are several new Windows messages, ranging from generalized application notification for the Plug and Play subsystem (WM_DEVICEBROADCAST) to the support for multiple keyboard layouts (WM_KBDLAYOUTCHANGE) required by fully international applications.
- Windows 95 presses into service some previously unused parts of existing data structures. New capabilities, such as automatic centering of a dialog box (the new DS_CENTER style bit), are supported. If your code "borrows" reserved or previously unused regions of Windows data structures, you may need to make some changes.[13]

Programming for Windows 95

Now that we've looked at some of the things you can't do on Windows 95, let's turn our attention to a more interesting topic: the new capabilities you can exploit as you create your next million-copy seller. The new features are accessible only by 32-bit applications,[14] so the first task is to port existing code to Win32. Together with all the new possibilities for Win32 applications come new rules and considerations. We'll look at those as we examine the new features.

There are many small enhancements to the Windows API, and we won't look at them all in any detail here. Reference works that analyze the new features will probably address this extensive topic. Checking the specification for all the APIs with the *Ex* suffix is one way to begin an investigation.[15]

Multitasking

As you saw in Chapter Four, the Windows 95 multitasking environment is dramatically different from the Windows 3.1 environment. If you've

13. To preserve compatibility, Windows 95 includes several internal version checks to preclude any attempt to interpret "old" data structures with new semantics. An executable with a version number of 3.1 or lower won't see any of this new behavior.

14. The DPMI challenge was met by a band of developers determined to prove that real mode code could use protected mode facilities. It'll be interesting to see whether someone comes up with a trapdoor for 16-bit Windows applications. But don't try this at home.

15. For example, there's even an *ExitWindowsEx()* API. Though you're unlikely to use it, on machines that support the feature, you can close down Windows *and* turn off the power.

programmed for other multitasking systems—Windows NT, UNIX, or OS/2—you're already familiar with some of the issues you'll have to deal with in the Windows 95 environment:

- Synchronization and sharing. You can never be sure that the operating system isn't going to preempt your application and take the processor away from you, so any use of shared objects, such as memory mapped files, must be synchronized with other applications' use. Assumptions about the timing of arriving window messages are also invalid for Win32 applications under Windows 95.
- Multithreading. Using additional execution threads to manage different windows or background operations such as file searching adds complexity to an application's code. But users will quickly come to expect such capabilities. The Windows 95 shell, for example, uses a separate thread for each visible window. If your application simply puts up the hourglass cursor during a lengthy operation and refuses to respond quickly to mouse clicks, it will suffer in comparison with applications that do allow the user to interrupt the operation or get on with something else.

A plethora of Win32 APIs are available to assist in thread synchronization. Many of them look similar to one another, but study of the details will reveal subtle but significant differences. Windows 95 supports all of the Win32 synchronization APIs. One group—made up of the *InterlockedIncrement()*, *InterlockedDecrement()*, and *InterlockedExchange()* APIs that allow manipulation of a single 32-bit word—was originally designed to help support Windows NT multiprocessor operations. Even though you'll never see Windows 95 controlling a multiprocessor system, the APIs are still valid on Windows 95.

Win32 synchronization primitives deal with *critical sections*, *events*, *mutexes*, and *semaphores*. Here's what's important about each:

A **critical section** is used by threads belonging to the same process. One thread declares a CRITICAL_SECTION variable and initializes it using the *InitializeCriticalSection()* API. Thereafter, any thread can call *EnterCriticalSection()* and *LeaveCriticalSection()* to protect code sequences in which it must be the only thread of the parent process allowed to run.

Any thread can create a named **event** object and obtain a handle to it using *CreateEvent()*. Other threads belonging to any process can obtain a handle to the same event by specifying the same event name. Any thread with a valid handle can then use the *SetEvent()*, *ResetEvent()*, or *PulseEvent()* API to signal an occurrence of the event. Threads waiting for the event are then free to continue execution, and multiple threads may become eligible to run when the event is signaled. These event APIs send a "one to many" signal, unlike the other synchronization APIs.

A **mutex** is a named object that you acquire a handle to by means of *CreateMutex()* or *OpenMutex()*. Again, any thread in any process can obtain a handle to the mutex if it knows its name.[16] Only a single thread can gain control of the mutex object—so this implements critical sections for cooperating processes. The *ReleaseMutex()* API relinquishes control of the object.

A **semaphore** object is controlled in a way similar to control of a mutex object by means of *CreateSemaphore()* and *OpenSemaphore()*. The difference between the two is that the semaphore can have a value. For example, if you have an application controlling an eight-line telephone dialer, you can set up a semaphore with a value of *8* to help manage line allocation. The first eight threads that ask for a line get one, and the next thread blocks, awaiting a line release by another thread, which uses the *ReleaseSemaphore()* API to increment the count.

All of the interprocess synchronization APIs use handles to identify the object in use, be it an event, a mutex, or a semaphore. When a thread wants to synchronize with another thread, it uses an API that allows it to wait for a single object (*WaitForSingleObject()* and *WaitForSingleObjectEx()*), or for one of possibly many objects (*WaitForMultipleObjects()* and *WaitForMultipleObjectsEx()*). The two multiple objects APIs can use an array of object handles supplied by the caller—plus a time-out—to simplify the synchronization procedure. The *MsgWaitForMultipleObjects()* API allows you to synchronize with any of these objects, or with a time-out, or with a Windows message arriving in the thread's input queue.

16. The *DuplicateHandle()* API allows you to pass the handle to another process. The receiving process doesn't have to know the name of the mutex object. This works with all the handle-based Win32 synchronization APIs.

Memory Management

In Chapter Four, we looked at the Win32 memory management APIs and at some aspects of their implementation. Remember:

- The need to lock resources and memory objects is gone. All objects exist within a huge 32-bit flat virtual address space. Assumptions about actual addresses of objects are probably wrong, and they're definitely nonportable. You have to address objects using only system-supplied handles or pointers.
- The system protects the private address space of each Win32 application. You can't get a valid pointer into some other Win32 application's address space.[17] To exchange information, cooperating applications must use the defined interprocess communication methods and synchronization APIs. The *WriteProcessMemory()* API is the only controlled way of modifying somebody else's address space, and this API is really meant only for use by debugging tools.
- You can't pass handles back and forth between Win32 applications except by using the *DuplicateHandle()* API. Just as actual memory pointers aren't valid in different processes, neither are handles. You have to use the *DuplicateHandle()* API to get a valid handle to pass to another process.
- Of the various shared memory allocation methods, using the *CreateFile()* and *MapViewOfFile()* APIs is the recommended method for sharing. The performance with this method is good, and the method is fully portable to Windows NT.

Plug and Play Support

Chapter Eight deals with the Plug and Play subsystem in detail. Much of the Windows 95 Plug and Play support involves device drivers, not applications, but there is one new Windows message specifically associated with Plug and Play operations. The WM_DEVICEBROADCAST message informs an application of changes to the system's hardware configuration. If your application or device driver is the controlling party, you can use the *BroadcastSystemMessage()* API to send this message.

17. But you can get a pointer into the shared region used by all of the Win16 applications and the 16-bit subsystem DLLs. Again, this is an artifact of the strict compatibility requirements for Windows 95.

Perhaps unusually, this particular message is important to both applications and system components, although the information the message sends is often of interest only to device drivers. At the application level, the device event code the message sends can provide, for example, information about the addition and removal of logical disk drives.[18] This would allow an application to respond sensibly to docking and undocking operations, for instance.

The Registry

The registry in Windows is a structured file that stores indexed information describing the host system's hardware, user preferences, and other configuration data. In Windows 3.1, the registry is used by applications to specify a limited amount of information, such as OLE document types.[19] In Windows NT, everything goes in the registry. Use of the registry in Windows 95 falls somewhere between these minimalist and all-embracing approaches.

The purpose of the registry is to reduce the proliferation of configuration files that can plague a Windows machine. In Windows 3.1, the CONFIG.SYS, AUTOEXEC.BAT, WIN.INI, and SYSTEM.INI files all contain information related to the system configuration. Some of the information is vital to the system's operation—specifying device drivers to load, for instance—and most of the remaining information describes other important aspects of the system's configuration. Add to these files the private .INI files set up by applications and the .GRP files used by the Program Manager, and it gets harder and harder to know where to look when diagnosing a problem or searching for a configuration setting.

Apart from the proliferation of these files in Windows 3.1, their integrity is a problem. Since the files contain plain text, the user can edit them directly, perhaps messing them up, and Windows has no way to figure out what might have happened. Incorporating all the configuration information into a registration database file and providing controlled access to it would preclude many of these potential problems.

18. This much was true in July 1994. It's clear that the device broadcast message could be extended to cover many different occurrences.

19. To be precise, Windows 3.1 supports a *registration database*, which the purist will argue is different from the Windows NT registry. It's a rather academic point.

Windows NT does away with all of the plain text files that Windows 3.1 uses and, in addition to the system's own use of the registry, allows applications to use the registry for storing private configuration data.

Windows 95 continues to process the configuration files you're familiar with—AUTOEXEC.BAT for example. Windows 95 also supports the registry. The principal user of the registry in Windows 95 is the Plug and Play subsystem, and all device-related information moves to the registry. Although this might seem to simply expand the file proliferation problem, you can use your own Windows 3.1 system as an example to measure the effect of putting this information in the registry. Count the number of lines in CONFIG.SYS, AUTOEXEC.BAT, WIN.INI, and SYSTEM.INI, subtract the lines that relate to hardware configuration, subtract other lines such as "BUFFERS=" that have no relevance under Windows 95, and you'll see that a lot of data disappears.[20] Although the development team would have preferred to adopt the registry mechanism in its entirety, the compatibility issues associated with upgrading the installed base of Windows 3.1 systems and their 16-bit applications were too great. The old style configuration files thus survive, but no doubt more and more use of the registry will be made in the future.

Figure 6-4 on the next page shows the arrangement of the registry database with its principal keys. Notice that the keys are hierarchically related, meaning that entire subtrees can be isolated and indexed with subkeys.

A particular software vendor might use the registry database to store application configuration information under the key HKEY_LOCAL_MACHINE\SOFTWARE\VENDOR\APPLICATION. In Figure 6-4, information about Exotic's spreadsheet application is registered this way. Typically the HKEY_LOCAL_MACHINE branch of the hierarchy describes non-user-specific information about the host system. The HARDWARE branch of this subtree is where the Windows 95 Plug and Play subsystem stores all of the system's hardware configuration information.

As you might expect, the registry APIs supported by Windows 95 don't include the security-related interfaces. Windows 95 does support

20. On my machine, CONFIG.SYS and SYSTEM.INI disappear altogether, and AUTOEXEC.BAT and WIN.INI shrink substantially.

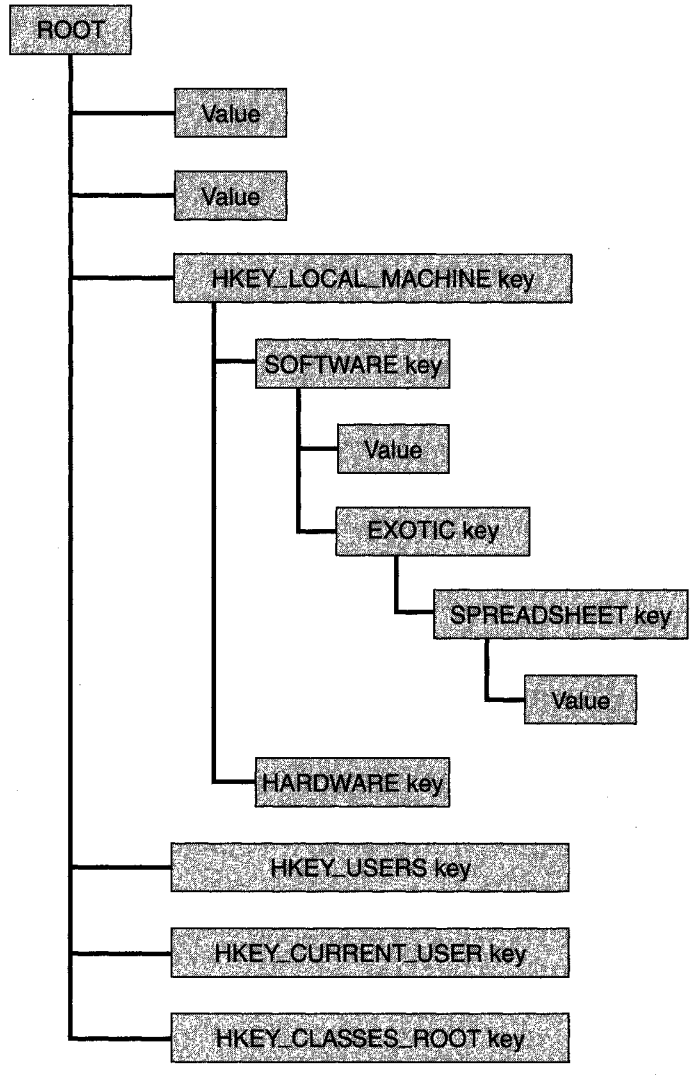


Figure 6-4.
Registry hierarchy in Windows 95, showing the principal keys.

a number of interfaces for VxDs that allow access to the registry, however. A subset of these interfaces is available for read-only access during the system's real mode initialization procedure.

The User Interface

In Chapter Five, we looked at the new visual elements of the Windows 95 user interface and at some of the Microsoft guidelines for making Win32 applications consistent with the shell and its behavior. Underlying the new appearance are many new and enhanced APIs. Two aspects of the new interface have an impact on existing applications:

- The minimum hardware supported by Windows 95 is a VGA display with 16 colors. The system itself operates internally with a palette of 256 colors, mapping the 256 to 16 if the hardware has only 16. The new 3-D appearance and the improved use of color mean that you should avoid hard coding colors, particularly for owner drawn items such as buttons. The *GetSysColor()* API helps you use the currently selected color palette within an application.
- The ability for the user to resize every visual element—scroll bar widths, caption bars, and the like—means that your code must not make assumptions about the size of standard items. User resizing coupled with the system's ability to change display resolution on the fly[21] plus the overall Plug and Play environment means that an application that truly exploits the Windows 95 capabilities must be able to react well to dynamic configuration changes.

The enhancements to the APIs you're familiar with cover many areas: menus, keyboard accelerators, icon management, and new capabilities that allow you to exploit the new visual appearance just as the shell does. Since this isn't an attempt to teach Windows 95 programming, we won't look at the details here. Suffice it to say that if Windows 95 is successful, its users will rapidly come to expect updated applications that exploit its new appearance and interface capabilities.

OLE

OLE has been the most widely promoted aspect of Microsoft's system software products over the last couple of years. Viewed initially as simply a "better DDE," OLE has evolved to become the cornerstone of

21. You'll receive a WM_DISPLAYCHANGED message both before and after this happens.

Microsoft's object-oriented system efforts. Windows 95 is the first operating system release that incorporates OLE as a standard function, although add-on libraries for Windows 3.1 have been available for developers to ship with their applications for some time. OLE's importance to developers is underlined by Microsoft's plans for the Cairo operating system to provide support for distributed object-based systems and an object-oriented filesystem—both of which are derived from the current OLE object model and compound file format. OLE today is a complex subsystem, but support for it within C++ class libraries continues to grow, somewhat simplifying the developer's task.

OLE has been dealt with extensively in other books. And doing justice to OLE would merit at least an entire chapter in this book. Nevertheless, it's important to at least look at some of the fundamental features of the technology.

OLE deals with collections of objects that make up *compound documents*. A compound document is a grouping of data prepared by several different applications. A letter prepared by a word processor, for example, might include a numeric table generated by a spreadsheet program. The Windows DDE capability offered limited facilities for using multiple applications to prepare and maintain such compound documents, but more often than not, the act of preparation involved simply copying a final version of the spreadsheet table, pasting it into the letter, and printing it. OLE aims to provide the framework wherein the user can prepare and maintain compound documents without losing any of the attributes of the data objects or precluding the possibility of manipulating the data objects in their original forms. This capability involves either maintaining a *link* in the compound document to the object in the original application, or *embedding* the data object directly within the document. In either case, when the user selects the data object, the originating application runs and provides the user with all of its data manipulation capabilities. The user can not only resize the spreadsheet as it sits in the document but also change the numbers and recalculate the contents.

An application that supports this architecture is called an *OLE server*, and an *OLE client* is any application that allows the inclusion of OLE objects within its supported document formats. Selecting an embedded object may cause the server to use an *in-place activation* technique whereby the server takes control of the client application's menus and of the redrawing of the screen area occupied by the data

object. There's no apparent switching to another application such as we're used to. The user just has a different set of operations available for that particular data object.

OLE-enabled applications support *drag and drop* operations, in which the user can select a graphical representation of a data object and deposit it on some other object, whereupon the target object does something useful with the data. The Windows 95 shell, for example, allows the user to drag an object onto the desktop and leave it there or to drag a document to a printer and have it be printed with no further interaction.

Since the client application in the printing drag and drop example would know nothing about how to print the document, it would rely on the *programmability* of the server application. Simply put, this means that the client can determine which application created the document and send the server application a print command together with the document data. The server will expose possibly many interfaces to its functions, and any client can call the server functions at will. A page layout program, for example, could call on the text justification function available in a word processor. No user actions would be required to make this happen—it's the OLE subsystem that initiates and controls the interactions among all of the components.

Microsoft calls the core of the OLE design the *component object model*, and under this broad heading lists all the programming interfaces, data structures, and protocols that control OLE operations. OLE relies completely on object-oriented programming techniques and in particular on C++. The written OLE specification is based entirely on C++ conventions. The implementation of OLE on Windows 95 requires the presence of several DLLs in the Windows directory.

The OLE *compound file format* specifies a storage mechanism for OLE objects and their associated data. Within one compound file, it's possible to create multiple *streams*—each of which can contain collections of logically separated objects. A compound file allows its contents to be indexed efficiently, and the index is permanently retained—just as a database index is. Windows 95 implements an OLE compound file by storing the streams and the index in a single disk file. To the operating system, the file is just a collection of bits. Only the OLE subsystem knows how to interpret the index and the data streams. All of that will change with Cairo, and the interfaces offered today by the OLE libraries will become part of the operating system proper. With Cairo,

Microsoft plans to offer a new *object filesystem* as the native storage format. Multiple stream support, indexing, and object storage and retrieval functions will be an inherent feature of this filesystem. Whereas today we have APIs that simply read and write data, Cairo will provide APIs to load and store entire compound documents.

Of course, Microsoft faces healthy competition from various quarters in its bid to establish OLE as the preferred object model for PC-based applications. But OLE is gradually establishing itself, and its ready availability in Windows 95 and Windows NT is a good way to approach the contest.

International Support

Over the years Microsoft has invested an enormous amount of effort in the process of translating its products for use in overseas markets. It isn't just a matter of translating program text and documentation. Issues of local currency and date formats and other cultural considerations abound. For the Far East and Middle East markets, the complex character sets and right to left parsing issues further increase the work required to make a software product truly international.

Windows 95 will represent Microsoft's largest investment yet in the internationalization of a product. The plan calls for Windows 95 to be released simultaneously in seven languages (English, French, German, Italian, Swedish, Spanish, and Dutch) and for many other language versions to follow in the subsequent six months. To achieve these goals, Microsoft has restructured its development methods. Whereas Windows 3.1 had been localized by having a variety of small teams modify the source code and carry out the language translation, the rule for Windows 95 has been no source modifications for localization purposes. Whatever changes were necessary for localization were done just once, in Redmond, and then the individual translation groups worked with binary resources only.

Apart from the effort it has invested in the localization project itself, Microsoft has also enhanced Windows 95 considerably for foreign language support. Among the design decisions that the team had to make, determining whether to use the Unicode character set, as Windows NT does, was one of the major ones. For compatibility and size reasons, Windows 95 is not a Unicode system, although a number of its components, such as OLE, use Unicode as their internal character

representation.[22] A range of Windows 95 features are aimed at simplifying the challenge of producing software that deals with many foreign languages:

- Support for multiple keyboard layouts, allowing dynamic switching between character sets. This means, for example, that more than one foreign language can be used and displayed within a single document.
- The so called *locale* APIs that handle issues such as string sorting, code page management, and localized date and time formats. A locale implies both a language dialect and a location.[23] So, for example, the issues associated with software for a multilingual country such as Switzerland can be correctly handled. Windows 95 allows you to control some 110 different locale items.
- Extensions to existing APIs, such as *MessageBoxEx()*, that allow an application to specify the language resources to be used for display of the text in buttons.

Structured Exception Handling

Although not specific to Windows, structured exception handling is a feature that Windows 95 supports. Together with operating system support, you have to have a compiler that supports the capability. One without the other won't do it. Windows NT with the Microsoft 32-bit C compiler was the first Microsoft environment to support structured exception handling, and now it's in Windows 95.

Structured exception handling allows the programmer to bring order and simplicity to the usually onerous chore of error handling. A condition such as an error code returned by a system API, or a memory fault caused by an invalid pointer, can be handled in one place rather than with code scattered throughout an application. Figure 6-5 shows

22. The principal problems were the growth in the size of the system's working set (remember that 4-MB requirement) if Unicode were to be used and the compatibility testing issues associated with modifying close to 500 individual APIs for Unicode support.

23. Even to the extent that American English can now be properly viewed as a dialect of the Queen's English!

an example of how you might handle errors the "old" way (including an "old" bug), and Figure 6-6 is the same code modified to use structured exception handling. Some of the obvious declarations have been omitted for brevity, and the code is a little artificial—but it serves to illustrate the technique.

This code fragment opens a file, reads the first word of the file to determine the size of the subsequent data record, allocates memory for

```
    // Open a file, read a word, allocate that many
    //  bytes, and read the data in. Scan for a pattern.
    //  Return TRUE if it's found, FALSE otherwise.
hFile = OpenFile(lpszName, lpOFbuff, wMode);
if (hFile == HFILE_ERROR)
    return FALSE;                   // Failed
wFlag = ReadFile(hFile, &lSize, 4, &lCount, NULL);
if ((bFlag == FALSE) || (lCount != 4))
    CloseHandle(hFile);             // Any error ignored
    return FALSE;                   // Failed
}
lpMem = VirtualAlloc(NULL, lSize, MEM_COMMIT, PAGE_READWRITE);
if (lpMem == NULL){
    CloseHandle(hFile);             // Any error ignored
    return FALSE;                   // Failed
}
bFlag = ReadFile(hFile, lpMem, lSize, &lCount, NULL);
if ((bFlag == FALSE) || (lCount != lSize)){
    // Deallocate memory, ignore any error.
    (void) VirtualFree(lpMem, lSize, MEM_DECOMMIT);
    (void) CloseHandle(hFile); // Any error ignored
    return FALSE;                   // Failed
}
lpScan = (DWORD *)lpMem;
    // Scan buffer: (bug: no bad pointer check
bFlag = FALSE;
while (TRUE)
    if (*lpScan++ == lPattern){
        bFlag = TRUE; break;
    }
    // Ignore errors on deallocation or closing.
(void) VirtualFree(lpMem, lSize, MEM_DECOMMIT);
(void) CloseHandle(hFile);
return bFlag;                       // Failed
```

Figure 6-5.
Handling errors the old way, without structured exception handling.

the data, and reads the data in. Errors can occur while the code tries to open the file, while it reads the file, while it tries to allocate the memory buffer, or when it searches the buffer for a given value—when the pointer steps past the end of the buffer. The code shown in Figure 6-5 laboriously tests for error conditions. The code in Figure 6-6 handles all possible errors by embracing the code in a single *try* block, defining an *except* block that will be called if any errors occur, and then cleaning everything up in a *finally* block that executes regardless of success or failure. Note that the *except* block in Figure 6-6 will execute in

```
try {
    hFile = OpenFile(lpszName, lpFBuff, nMode);
    (void) ReadFile(hFile, &lSize, 4, &lCount, NULL);
    if (lCount != 4)    // Bad file format:
                        //  raise private exception
        RaiseException(BAD_FILE_DATA,
                       EXCEPTION_NONCONTINUABLE, 0, NULL);
    lpMem = VirtualAlloc(NULL, lSize, MEM_COMMIT,
                         PAGE_READWRITE);
    (void) ReadFile(hFile, lpMem, lSize, &lCount, NULL);
    if (lCount != lSize)   // Bad file format
        RaiseException(BAD_FILE_DATA,
                       EXCEPTION_NONCONTINUABLE, 0, NULL);
    lpScan = (DWORD *)lpMem;
        // Bug still present
    bFlag = FALSE;
    while (TRUE)
        if (*lpScan++ == lPattern) {
            bFlag = TRUE; break;
        }
}
except (EXCEPTION_EXECUTE_HANDLER) {
    // Do the exception analysis here.
    bFlag = FALSE;      // Set error indication;
                        //  cannot return from here
}
finally {
    (void) VirtualFree(lpMem, lSize, MEM_DECOMMIT);
    (void) CloseHandle(hFile);
}
return bFlag;           // Return the result
```

Figure 6-6.
Using structured exception handling.

the event of a memory access fault when the code scans past the end of the allocated memory buffer (with the pattern not found). Neither example tests for this condition, and in the first case you'd get a program failure with little useful qualifying information.

Exception handlers are frame based, meaning that their scopes nest just as declaration scopes do, so it's possible to handle errors on either a global or a local basis. There are also facilities for specifying the context in which the exception is handled.[24] The structured exception handling feature also allows a program to initiate an exception (the *RaiseException()* API) and specifies the protocol for interacting with a debugging tool if one is in use. Within an *except* block, you can determine the cause of an exception so that you can carry out appropriate error recovery. You shouldn't replace every error test in your code with an exception sequence, but it is a great way to manage a multitude of possible error conditions diligently and efficiently. After all, how many times do you test for every possible error in your code?

The Graphics Device Interface

GDI is the heart of the Windows graphics capabilities. All of the drawing functions for lines and shapes are in GDI as well as the color management and font handling functions. Many aspects of Windows performance are tied closely to GDI performance, and a lot of the GDI code is handcrafted 386 assembly language. At the application level, Windows provides logical objects known as *device contexts* (DCs) that describe the current state of a particular GDI drawing target. A DC can describe any output device or representation of a device. An application will obtain a DC for printer output or for completely memory-based operations, for example. Applications manage DCs by means of Win32 APIs only. The actual DC data structure is always hidden from the application. At any instant a DC contains information about objects such as the current pen (for drawing lines), the current brush (for filling regions), the color selection, and the location and dimensions of the logical drawing target.

The key to the use of Windows and Windows applications on a widely disparate range of target hardware is the device independence

24. Reminiscent of, but much better than, the C language *setjmp()/longjmp()* facility.

embodied in the Windows API. An application uses DCs and other logical objects when calling GDI functions. It never writes data directly to an output device. GDI itself manages the process of transforming the data into a format suitable for use by a particular device driver, and the driver handles the task of placing a representation of the request on the output device. For example, an application may call the system asking for its main window to be repainted. During the repainting operation, among many other requests, GDI may tell the driver "on the screen draw a one-pixel-wide black line from position (0, 48) to position (639, 48)." If the device—a dot matrix printer, say—can't perform operations such as line drawing, GDI will break the request down into simpler operations. The device driver will receive a series of calls telling it to draw individual dots, for example. This architecture frees applications from device-dependent problems and allows Windows to make use of even the simplest hardware as an output device.

With this device-independent capability come several problems. In addition to simply choosing and managing an appropriate device-independent representation of all the graphics objects, you need to have a plethora of device drivers available to interface GDI to the target hardware. Issues such as handling complex fonts through a range of point sizes and then being able to draw the font legibly on both a 1024 by 768 pixel display screen and a simple dot matrix printer involve many complex algorithms and a lot of very clever code.

Over successive releases of Windows, the capabilities of GDI have improved considerably, and the underlying structure of the system has adapted to the experience gained from earlier versions and to the prevailing market forces. The vast majority of Windows users nowadays tend to have fairly capable hardware: VGA displays and laser or high-resolution dot matrix printers. The hardware will probably get even more powerful, with higher resolution and color-capable devices abounding. It's therefore important to get the best possible performance out of a few core components rather than expend effort on hundreds of device drivers, each with a limited installed base. It has also been important to look ahead at the likely effects of hardware trends. Two of the major changes in the Windows 95 GDI subsystem reflect hardware trends: the *device-independent bitmap* (DIB) engine and the *image color matching* (ICM) subsystem.

Windows 3.1 successfully introduced the concept of the *universal printer driver*—a device driver that does much of the work for all the other system printer drivers. The so called *printer mini-drivers* support

only the hardware-specific operations of a printer and rely on the universal driver for most printing-related functions. This allocation of responsibility allowed Microsoft to invest heavily in a high-performance, high-quality universal printer driver and in some good example mini-drivers for devices such as the Hewlett-Packard LaserJet. From the printer manufacturer's perspective, Windows printer driver development became a much simpler and much less error prone project.

Windows 95 takes up this design concept by incorporating the DIB engine and a *display mini-driver* capability. If the display hardware matches what the DIB engine can do, what was once a very complex, performance-sensitive development effort is considerably simplified.[25] Write a display mini-driver, and rely on the DIB engine as (in Microsoft's phrase) "the world's fastest flat frame buffer" display driver. The DIB engine design also recognizes the level of effort that hardware manufacturers now put into hardware assists for Windows-based systems. If you have hardware acceleration or other capabilities, the display mini-driver can use these instead of calling the DIB engine.

Image color matching is a new capability that addresses device-independence issues for applications that deal with color, such as photo retouching applications. Although color has always been part of Windows, earlier releases didn't have to worry too much about the issue since color-capable peripherals were relatively rare. But now that the price of good color scanners and color printers has fallen to the $1,000 range, Windows has to take careful note of color management.

Here are the other improvements to GDI in Windows 95:

- Performance. A lot of code has been tuned, and some important components have been converted to 32-bit code.
- Relaxation of resource limitations. In parallel with what's been done to the User subsystem, many of GDI's resource limits have been raised significantly.
- Win32 support. Windows 95 fully supports many graphics APIs unavailable in Windows 3.1.
- TrueType enhancements.

25. One simple code count shows the VGA display driver in Windows 3.1 to be over 41,000 lines of assembler (for a 16-color–only display). In Windows 95, it's only about 5000 lines for the full 256-color driver.

- Metafile support enhancements compatible with Windows NT's metafile support.
- Printing subsystem enhancements, including bi-directional printer support and a new 32-bit print spooler.

GDI Architecture

Figure 6-7 illustrates the major components of the GDI subsystem. It also shows the breakdown between 16-bit and 32-bit code modules—with one caveat: the DIB engine is actually 32-bit code running with a 16-bit (segmented) view of system memory—so the code makes use of the fast 386 instructions for memory move operations, for example. There's considerable trickery involved in efficient address manipulation, but it means that existing 16-bit applications can realize the performance improvements of the new DIB engine and that the engine itself can call into the 16-bit GDI code with no additional overhead. If the DIB engine were placed on the 32-bit side of the fence, either the

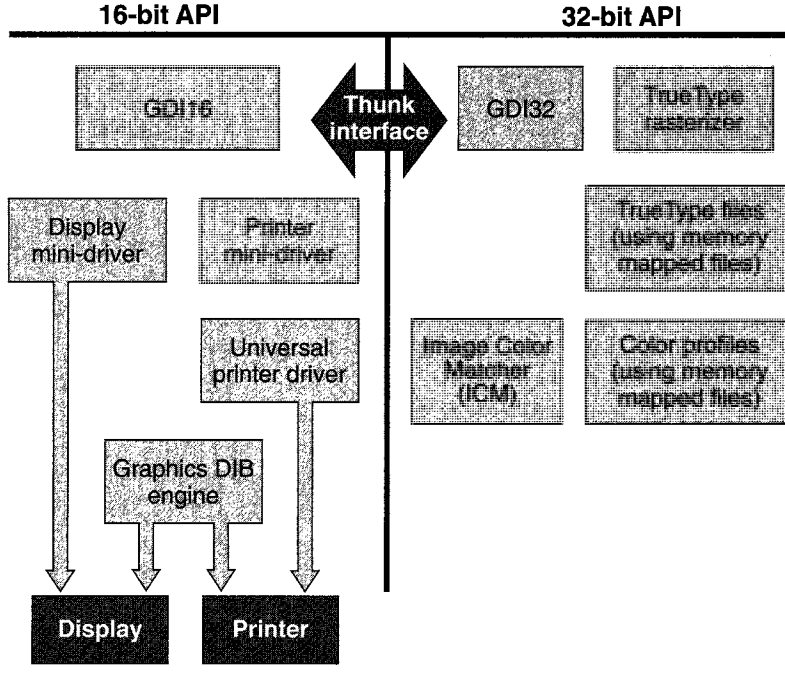


Figure 6-7.
The components of GDI in Windows 95.

32-bit GDI module would have to replicate much of the GDI functionality, or the DIB engine would incur lots of thunk overhead calling back to the 16-bit side.

Before looking in detail at the new DIB engine and the ICM subsystem, let's review the smaller improvements in the Windows 95 GDI.

Performance Improvements

The performance of the GDI subsystem is critical to the performance of Windows. Many benchmarks of Windows 3.1 tend to focus attention on video performance. Although video performance is only one element of the overall performance of the system, it's certainly a huge factor in perceived performance.

The Windows GDI code has been worked on for a sufficiently long time that there really aren't any huge undiscovered performance gains to be made. But Windows 95 includes quite a few incremental improvements:

- The new DIB engine is handcrafted assembler. The effort invested in this will improve the performance of many video display drivers as well as the print subsystem.
- The TrueType rasterizer is the component responsible for turning a description of a font into the actual image you see on the screen or on the printed page. The Windows 95 rasterizer is new 32-bit code.
- The print subsystem spools print metafiles, reducing the amount of data movement and hence speeding up the print process. The print spooler itself is new 32-bit code that can run as a true background process.
- A lot of new 32-bit code in key components makes use of the improved instructions available on the 386 processor. Also the duplication of some GDI components in 16-bit and 32-bit code avoids thunk overhead.

Limit Expansion

Along with the move partway to a 32-bit subsystem comes access to the 32-bit memory pools used by Windows 95. Under Windows 3.1, the GDI subsystem allocated all resources from a single 64K heap—which limited the total number of available resources significantly on systems that were capable of running several applications at once.

In Windows 95, GDI still keeps many logical objects in a heap limited to 64K. The data structures that describe brushes, pens, and bitmap headers, for example, stay in this smaller heap. Display context structures also remain in this pool. However, GDI now allocates the objects that can really eat up space from a separate, 32-bit memory pool. GDI regions, font management structures, and physical objects all move to this pool, which considerably reduces the pressure on the 64K heap. For example, the collection of rectangles used to describe an elliptical region can consume up to 45K. Decisions over which objects to move out of the 64K heap were also influenced by performance considerations. Since both 16-bit and 32-bit code has to manipulate the structures, the designers had to be careful not to incur too many selector loads when switching between the different heap areas.

New Graphics Features

Windows 95 incorporates almost all of the more advanced graphics APIs defined by Win32. Their inclusion increases the suitability of Windows for use as an application platform by graphics-intensive applications. The new APIs encompass

- Support for *paths,* allowing an application to describe a complex arrangement of geometric shapes that GDI will outline and fill with a single function call
- *Bézier curve* drawing, in which an application describes a curve using a series of discrete points and GDI figures out how to draw the curve

Applications such as high-end drawing packages and CAD products have to concern themselves with the very accurate representation of geometric objects. One of the differences between Windows 95 and Windows NT is in the drawing algorithms that define the pixels used when an application draws lines or fills shapes. Internally, an application can draw anywhere within the 16-bit coordinate space (−32,767 to +32,767 in both the x and y directions). GDI may have to scale this image dramatically to allow its display on a 640 by 480 pixel screen and, regardless of scaling issues, drawing a diagonal line on a video screen is always problematic. Essentially, GDI and the display driver have to figure out between them which pixels become black and which stay white. For most of us (and most applications), the differences between lines

drawn according to the two algorithms won't be discernible. There are similar subtle differences between the ways the two GDI subsystems fill shapes on the screen. The algorithms differ as they determine which pixels to include or exclude around the edge of the shape.

TrueType

The new TrueType rasterizer is implemented in C. It's an adaptation of the C++ module developed for Windows NT.[26] The new code also implements an improved mathematical representation of a font, using 32-bit fixed point arithmetic with a 26-bit fractional part. Windows 3.1 uses a 16-bit representation with a 10-bit fraction. This led to some rounding error problems (leading to reduced fidelity on high resolution devices) and difficulties in handling complex characters such as those in the Chinese language (the Han characters).

The rasterizer now uses memory mapped files to access font description files (all those .TTF files in your Windows system directory), and the associated .FOT files are gone. During the system boot process a private record of an installed font is written to disk and used during the next boot. This improves the speed of system startup considerably if you have a lot of fonts installed.

Metafile Support

Metafiles contain sequences of graphics operations written in a device-independent format. An application can obtain a device context to a metafile and draw a picture using the DC. GDI generates the metafile records that correspond to the GDI function calls made by the application. Metafiles can be reprocessed with the drawing output directed toward any capable device. The recorded picture will appear with the original sizing, proportions, and colors intact.

Windows 95 adds support for the enhanced metafiles defined for Win32, including limited support for world transforms (scaling operations only). There are some Win32-generated metafile records that Windows 95 won't understand, so it skips them when reading the metafile. This means that a metafile generated on a Windows NT system using the full range of graphics capabilities can't be completely reproduced on a Windows 95 system.

26. The Windows NT operating system code uses C++ extensively. There's none in the Windows 95 operating system.

Image Color Matching

The problem of producing a completely device-independent color capability for Windows remains an intractable one. There doesn't yet exist a recognized solution to the problem—for any general purpose computer system. Accurate color reproduction is the subject of many research projects, and a number of international standards try to solve subsets of the problem. Interestingly, all color standards in use today are derived from a 1931 definition known as the CIEXYZ standard. Apart from the fact that color reproduction involves issues of human perception, the basic problem is that even if you can define a completely adequate internal color representation system, no two devices will reproduce a given color identically. Thus, a "red" on the printed page will look different on the screen, and many colors that you can choose for your latest Van Gogh knockoff on screen can't be accurately matched by the colors your printer can produce. Given the inability of a device to produce a particular color, what do you do? Adjust that color to the nearest one available on the output device? Or adjust every color in the image in an attempt to maintain the original contrast? It doesn't seem likely that anyone will ever solve the problem to the complete satisfaction of every expert.

Color management systems that do exist today are built around specific hardware, so the controlling software knows what colors are available and what transformations it must use to render accurate color output. This of course runs counter to the Windows philosophy of always maintaining device independence. Yet the need for a good color management system is apparent. For a few thousand dollars, you can set yourself up with a very high quality color production system, and the prices will no doubt fall further. Thus, the Windows designers were faced with the challenge of integrating a color management system that meets the nonexpert needs (and budgets) of most of us while still supporting the stringent requirements posed by professionals in magazine publishing and photographic reproduction.

Image color matching (ICM) is Microsoft's name for the solution incorporated into Windows 95:

- ICM defines a *logical color space* for Windows that is defined in terms of the RGB (red, green, blue) triplets already used in Windows 3.1. The use of the existing RGB mechanism is really

a convenient implementation detail. The logical color space is actually calibrated with reference to the CIEXYZ standard.

- ICM uses a *color profile* that defines the color capabilities of a particular device. Manufacturers of color output peripherals can ship a color profile with their devices, much as they might ship a Windows device driver today. If a device has no associated color profile, the system chooses a sensible default profile.
- The color profile allows the ICM to build a *color transform* that defines how to map colors from the logical color space to the colors reproducible on the output device. For an input device such as a scanner, ICM uses the profile to transform the device colors to the logical color space.
- ICM thus allows device drivers and the system itself to perform color matching and color transformation operations in support of scanning or reproducing images involving a specific device. ICM aims to be consistent—giving you predictable results each time you scan, display, or print an image.
- ICM is implemented as a replaceable DLL, and it's possible to load more than one ICM at a time.[27] This means that for environments with different color management needs the system's default processing can be replaced or circumvented.
- Windows 95 adds support for the CMYK color standard that's widely used in applications that produce color separations for printing and publishing. If an application chooses CMYK as its color space, Windows stays out of the way and the application can pass color coordinates to the device driver without further transformation by the ICM.

Microsoft also realized early on that there were people who knew a lot more about color management than they did. The specification and development of the Windows 95 ICM was done in conjunction with Eastman Kodak, a company that does indeed know quite a lot about color. The default ICM DLL planned for inclusion with Windows 95 was written largely by Kodak.

27. Loading a new ICM is under application control. Two new APIs—*LoadImageColorMatcher()* and *FreeImageColorMatcher()*—manage the procedure.

Color Profiles

Microsoft will publish the format of a color profile in the Windows 95 SDK and DDK products. The definition will describe both the file and in-memory formats for color profiles. No doubt some standard profiles will be included with Windows 95 when it ships—just as you get most of your printer drivers "in the box" today. The contents of a color profile have been determined by efforts involving several different companies, and it's freely acknowledged that there are application areas that will need further extensions of the information embodied in a color profile. But for most applications, these color profiles are sufficient.[28]

As you'd expect, color profiles will be available for scanner devices, display screens, and color printers. The profile definition also enables the specification of profiles that describe abstract devices (allowing color effects) and color space conversion (from the internal logical color space to a different standard) and the specification of *device link* profiles. A device link profile caters to a system with a fixed configuration, allowing the color transformations to be fine tuned so that, for example, the particular "red" generated by your Hewlett-Packard ScanJet becomes exactly this "red" on your HP DeskJet printer.

Don't imagine that you'll be generating color profiles the same way you change your desktop colors with the Windows Control Panel, though. Color profiles are real science and may involve device calibration, temperature correction, and the handling of different paper and ink types, among other complexities.

Communicating Color Information

Figure 6-8 on the next page illustrates the flow of color information among the various components in the system. The color information communicated among the components is always expressed in either RGB or CMYK values, or in some transformation of these values according to the way the application has defined its color space.

At the application level, GDI provides several new APIs that allow a specific color space to be defined and manipulated.[29] An application uses a device-independent bitmap (DIB) to store an image, and the

28. If you don't already believe that color management is a tough problem, note the way in which the ICM designers acknowledged the difficulty. They listed one of their goals as specifying a system that's "simple enough to implement in our lifetimes."

29. If you're interested in the details, look for all the ICM-related APIs—those that have the string *Color* somewhere (!) in their names.

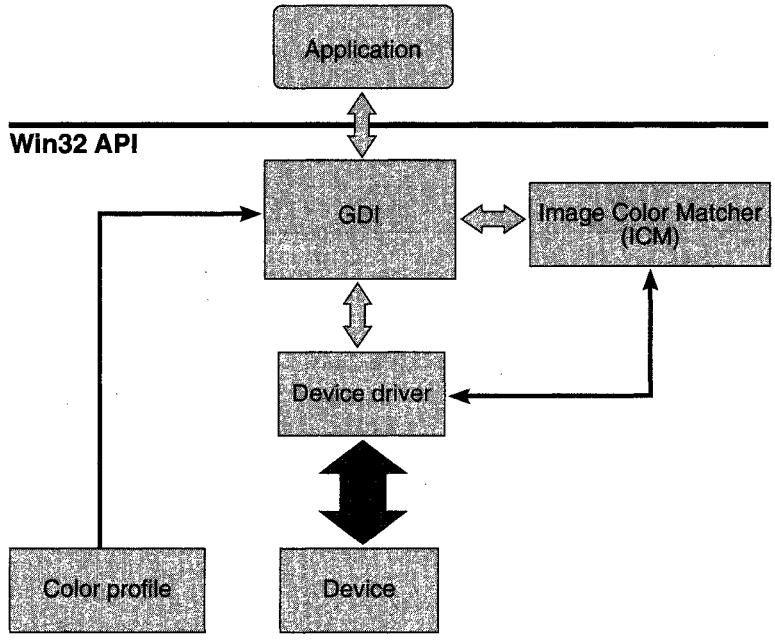


Figure 6-8.
Color information handling within the system.

color matching APIs operate directly on the bitmap. The DIB structure itself has been extended to incorporate color information, and, as with other device-related operations, color manipulation is specific to each Windows device context.

The Display Subsystem

Although Windows allows only a single system display device to be active, several different software components are involved in controlling the display. Figure 6-9 illustrates most of these components, together with the boundaries between them—the API layer and ring zero components vs. ring three components. The example in Figure 6-9 assumes a configuration that uses the new device-independent bitmap engine. The DIB engine assumes a major role in the control of the video display under Windows 95. In a configuration that doesn't use the DIB engine, the engine and the associated display mini-driver won't be present, and the system components such as GDI interact with a single

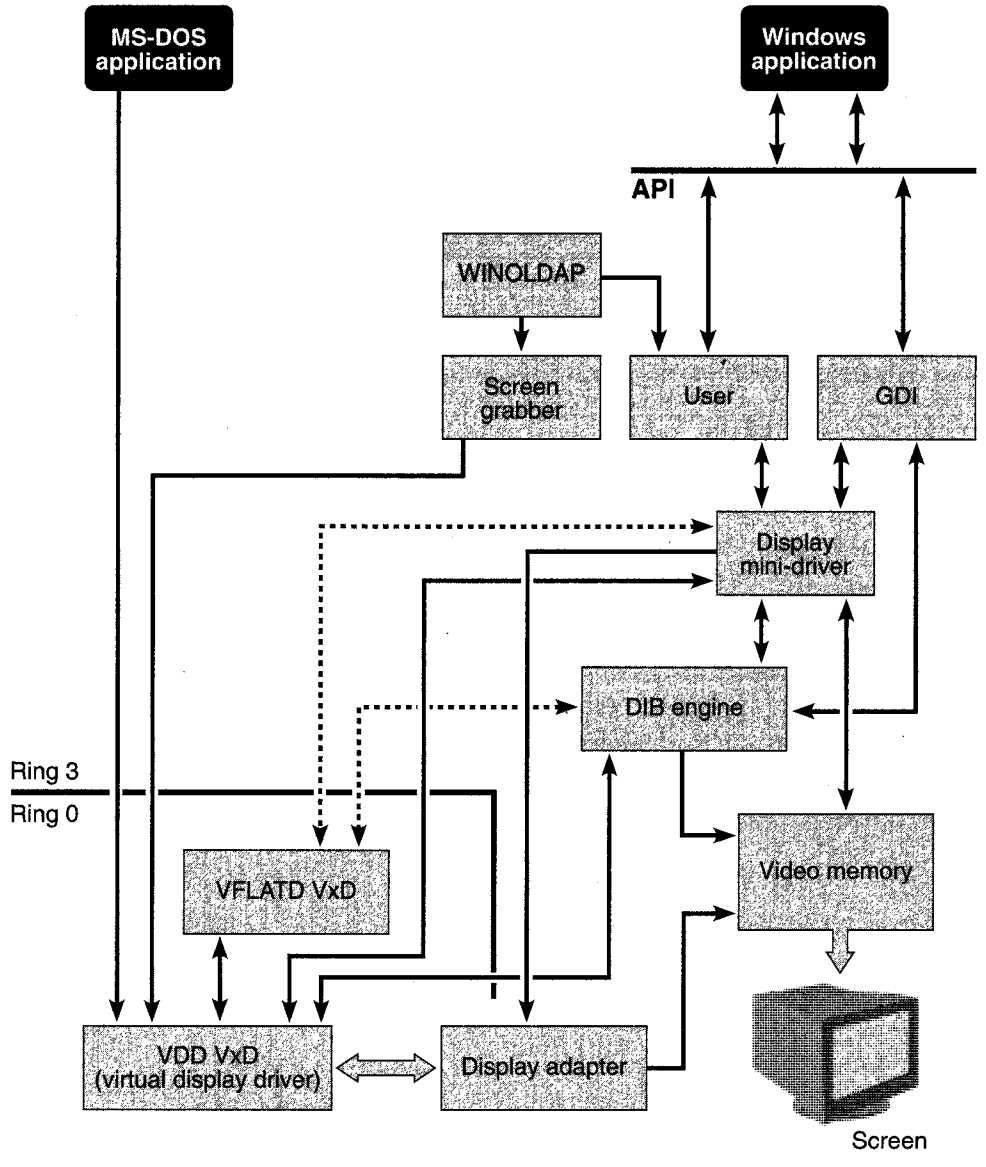


Figure 6-9.
Display subsystem components in an example configuration.

display driver module. That's essentially how Windows 3.1 works today, but in a very large percentage of Windows systems, the video hardware will be appropriate for use of the new DIB engine and the display mini-

driver architecture introduced with Windows 95. For most purposes, you can think of the DIB engine and the mini-driver as a single display driver. Windows always assumes that the video display is directly addressable as a memory region. Display adapters that don't allow this aren't usable by Windows for graphics operations.

Video system performance is critical to Windows, so, in terms of the length of instruction sequences, bringing the video memory and the video display adapter as close as possible to GDI is an overriding consideration. The device-independent nature of GDI means that it has to go through a device driver to get to the hardware and, in fact, two device drivers are involved. One is the VxD responsible for virtualizing the video hardware and controlling the switching of the screen between different virtual machines. (This is the VDD in Figure 6-9.) The other driver is a ring three DLL that always runs in the context of the system virtual machine. (This is the combination of the display mini-driver and the DIB engine in Figure 6-9.) So when the Windows desktop is on the screen (meaning that any MS-DOS applications are either not running or running in the background), the path from a Windows application to the screen is fairly efficient: a call to one of the Windows system DLLs, which in turn calls the display driver. No ring transition is involved, and the display driver has direct access to the video memory.

If Windows needs to initiate a hardware control operation—for example, to switch the screen resolution—it does rely on the display driver VxD. Normally, the ring three display driver will use the INT 10 video services interrupt to do this. The INT causes a fault, which initiates a ring transition. The kernel unravels the cause of the fault and hands control to the display driver VxD. Typically the display VxD will be the only component that mucks with the display adapter hardware.

The *grabber* module in Windows 95 is the same as in Windows 3.1. To support MS-DOS applications, the system's WINOLDAP module relies on a screen grabber for the purpose of saving and restoring the state of the video hardware and the video memory. The grabber has to match the display hardware type, so the grabber, the display VxD, and the display mini-driver are developed in concert. The VxD services used by the grabber include functions for copying data back and forth between video memory and a memory buffer, and various synchronization primitives that assist in critical section management and switching between virtual machines.

The DIB Engine

In Windows, a *bitmap* is a memory-based representation of a completed sequence of GDI operations. The resulting object is suitable for immediate display on a compatible output device, and, in the case of a device-independent bitmap, minimal additional processing will prepare the object for output to a different device. Bitmaps appear in files (the desktop wallpaper, for example), as application resources (the pictures on toolbar buttons, for example), and in main memory, where applications and device drivers can build and manipulate them directly. The entire Windows desktop display is itself a large bitmap, and the code that deals with updating the screen is critical to the system's performance.

The Windows 95 DIB engine recognizes the current state of display hardware by implementing a bitmap management capability that deals very efficiently with color flat frame buffer devices. In hardware terms, this would mean that the output device provides a large linear memory space with each screen pixel directly addressable as a memory location. Associated with each pixel is a color, represented by a number of bits. The DIB engine handles 1, 4, 8, 16, or 24 bits per pixel color, giving it a range from simple monochrome displays to high-end output devices with the ability to display millions of colors.

The DIB engine architecture assumes that it can set a particular pixel to a particular color by simply storing the appropriate number of bits in the correct memory location in the device's frame buffer. If the hardware doesn't have a frame buffer, the DIB engine is usable only for assistance in manipulating memory resident bitmaps: it doesn't try to allocate some huge chunk of memory and pretend it's the display device. Although the principal use of the DIB engine is for managing the video display, its bitmap manipulation capabilities lend themselves to other operations as well. Printer drivers can call the DIB engine for assistance when preparing a page, and GDI can use the DIB engine for operations on memory resident bitmaps.

Associated with the DIB engine is a display mini-driver called by GDI. This driver is still responsible for managing hardware-dependent operations in collaboration with the display driver VxD. GDI never calls the DIB engine directly, and, ordinarily, the DIB engine will rely on the mini-driver for hardware-dependent operations.[30] Also, if the

30. Among other enhancements such as color cursors and 32-bit color devices, Microsoft is already thinking about extending the use of the DIB engine so that GDI can indeed call it directly.

display adapter has additional capabilities, such as hardware acceleration for text output, the mini-driver is responsible for directly using these features and the DIB engine won't be called to perform that function. As part of its effort to get complementary hardware designed for Windows, Microsoft has been lobbying display adapter manufacturers to build devices with flat frame buffers, local bus video memory, and hardware acceleration for text output and bit blt operations.

Both the display mini-driver and the DIB engine are dynamically loadable libraries. Display drivers that rely on the DIB engine will cause it to be loaded during initialization. If the display driver doesn't use the DIB engine, it won't be loaded. The bitmap memory manipulated by the DIB engine is shared with GDI. For performance reasons, there's an attempt to minimize any back and forth copying of bitmaps.[31] The design of the DIB engine also tries to recognize the needs of multimedia applications with very high speed video data transfer requirements.

The Display Mini-Driver

The display mini-driver uses two major data structures to interact with the DIB engine and GDI. The GDIINFO structure is central to all of GDI's device-related operations. The structure defines, for example, the capabilities of the device in terms of its ability to draw lines, circles, text, and so forth. Many calls between GDI and its device drivers pass a pointer to the appropriate GDIINFO structure as one of the parameters. Information common to all devices is collected in the GDIINFO structure.

The other data structure is the DIBENGINE shown in Figure 6-10. Every GDIINFO structure specifies the size of the device descriptor structure associated with the device. Usually referred to as the PDEVICE structure, this data structure is entirely device dependent. Its size and contents vary according to the type of the device. For a display mini-driver, the PDEVICE structure is a DIBENGINE structure. Taken together, the GDIINFO structure and the DIBENGINE structure describe everything GDI needs to know about a display device that uses the DIB engine.

31. There's an analogous *CreateDIBSection()* API in Windows 95 that allows an application to reserve a directly addressable memory region for a bitmap that it shares with GDI.

```
struct DIBENGINE {

WORD        deType;             // 'DI' when GDI calls;
                                //  0 or ScreenSelector
                                //  when mini driver calls
WORD        deWidth;            // Width of DIB in pixels
WORD        deHeight;           // Height of DIB in pixels
WORD        deWidthBytes;       // No. of bytes per scan line
BYTE        dePlanes;           // No. of planes in bitmap
BYTE        deBitsPixel;        // No. of bits per pixel
DWORD       deReserved1;        // Reserved
DWORD       deDeltaScan;        // Displacement to next scan
LPBYTE      deLpPDevice;        // Pointer to associated
                                //  PDevice
WORD        deBitsOffset;       // Offset to DIB
WORD        deBitsSelector;     // Selector to DIB
WORD        deFlags;            // Additional flags
WORD        deVersion;          // Version 0x0400
LPBITMAPINFO deBitmapInfo;      // Pointer to BitmapInfo
                                //  header
void (FAR *deCursorExclude)();  // Cursor Exclude callback
void (FAR *deCursorUnexclude)(); // Cursor Unexclude callback
DWORD       deReserved2;        // Reserved

};
```

Figure 6-10.
The DIBENGINE data structure.

Bank-Switched Video Adapters

Another important component of the DIB engine architecture is a VxD called VFLATD, the flat frame buffer VxD. This VxD caters to display adapters that possess large amounts of video memory but have to use a memory window to switch back and forth between different 64K blocks of it.[32] The VFLATD VxD will manage up to a 1-MB logical frame buffer. The display mini-driver initially contains the code for switching the physical frame buffer to a different region of the logical frame buffer. When the mini-driver calls VFLATD to register this bank-switching code, the VxD actually copies the code into its own memory. Whenever the video memory window needs to be moved, VFLATD simply

32. If you remember expanded memory, that's exactly what this is like.

executes the switching code by running through it—not even a function call to get in its way as it comes steaming through!

Providing the bank-switching support as a standard part of the system (and making sure it runs as fast as possible) makes the mini-driver solution applicable to a much broader range of display adapters, so the likelihood of your system's using the DIB engine is pretty high.

Interfacing with the DIB Engine

When Windows 95 first loads a display mini-driver and calls the driver's DLL initialization routine, the driver simply collects information about its own configuration from the SYSTEM.INI file. Later on in the system's initialization process, GDI calls the driver's *Enable* interface twice. The first time through, the driver calls *DIB_Enable()*. The DIB engine hands back a pointer to an appropriate GDIINFO structure. The driver fills in some of the device-dependent fields (for example, the number of bits per pixel) and returns the GDIINFO structure pointer to GDI. The second call to *Enable* is where the rest of the initialization work gets done, including calling the display VxD to set the hardware into the correct graphics mode (using an INT 10) and if necessary handing the bank-switching code to the VFLATD VxD.

Once all the initialization is over, GDI, the mini-driver, the DIB engine, VFLATD, and the display VxD are all hooked together and ready to actually put something on the screen. The display mini-driver provides a standard set of about 30 or so interfaces that allow GDI to interact with the driver. Many of these functions are the same as those defined for existing Windows 3.1 display drivers, such as those for managing the cursor. All of them are exported entry points from the driver DLL. Several functions simply accept the call from GDI and hand it directly to the DIB engine. For example, GDI will call the driver's *BitmapBits()* function whenever an application creates or copies a bitmap. The mini-driver can turn around and call the DIB engine's *DIB_BitmapBits()* entry point with no transformation of parameters or, indeed, any other processing.

Management of the cursor is handled largely by the mini-driver, and, as with Windows 3.1 display drivers, the mini-driver must define the set of standard cursor resources used by GDI. This includes objects such as the standard arrow pointer, the I-beam cursor used in text fields, and the cursor we all hope we'll see a lot less of, the hourglass.

The Printing Subsystem

Much of the Windows 95 printing subsystem architecture (and indeed a lot of the code) is shared with Windows NT, so much of the new terminology and the new components of the print subsystem will be familiar to you if you've studied Windows NT. Apart from the new Image Color Matching capability, Windows 95 doesn't introduce any dramatic changes into this printing architecture, although across the board there are a number of significant improvements over the printing subsystem in Windows 3.1:

- A new spooler, implemented as a fully preemptive Win32 application. Print spooling can thus be a true background activity under Windows 95.
- Support for PostScript Level 2—the version applicable to color output devices.
- Bi-directional communication with the printer, which enables good Plug and Play support and the possibility of other enhancements.[33]
- Use of the new device-independent bitmap engine for high-performance bitmap manipulation.
- A new "quality of service" mechanism that allows the system to manage the simultaneous operation of more than one printer driver for a particular device.
- Improvements in the tools used for developing printer mini-drivers.

The Windows 95 printing system also expands the use of the printing APIs in preference to the printer escape functions used in Windows 3.0. An escape function (generated using the now-obsolete *Escape()* API) allowed an application to make a direct request to the printer driver. Windows 3.1 and Windows NT have replaced more and more of these escapes with APIs, and the recommendation now is to always use the API.[34]

33. A sample of these enhancements is already available in Microsoft's Windows Printing System product for Hewlett-Packard LaserJet printers.

34. The documentation for the *Escape()* API describes the details.

Printing Architecture

Three groups of components collaborate to print pages under Windows 95:

- GDI and its supporting modules, such as the DIB engine and the printer driver, which are responsible for translating drawing primitives issued by applications into a data stream suitable for the target printer.
- The local print processors and the print spooler that accepts the data stream and either writes it to a local disk file for subsequent printing or hands it to a local printer *monitor* for output to the physical printer.
- The despooler process and the *print request router* (PRR) that takes a print job and dispatches it to the correct target printer. This printer may be either a locally connected device or a network-attached printer.

Figure 6-11 illustrates these components and their interaction. In Chapter Nine, we'll look in more detail at the PRR and at the management of network printing. Essentially the PRR determines where a print job is headed and passes it to either the local printing system or the appropriate network subsystem for printing on a remote machine.

The Printing Process

An application produces output for a printer as it does for any other graphics device: it asks GDI for an appropriate device context and then draws its output using the DC. Obtaining the DC is a little different because the application must use the *CreateDC()* API, naming a target printer rather than simply requesting one of the available display device contexts maintained by Windows. Once it has the DC, the application uses the *StartDoc()* and *EndDoc()* APIs to identify the beginning and end of a discrete print job. Within a single job, the *StartPage()* and *EndPage()* APIs identify page breaks within the document.

Within the system, GDI, the printer driver, the DIB engine, and the local spooler combine to generate a disk file containing the data destined for the printer and an information file used to describe this *print job*. Both Windows 3.1 and Windows NT use a series of *journal records* as the basis for the print data file. The despooler is responsible

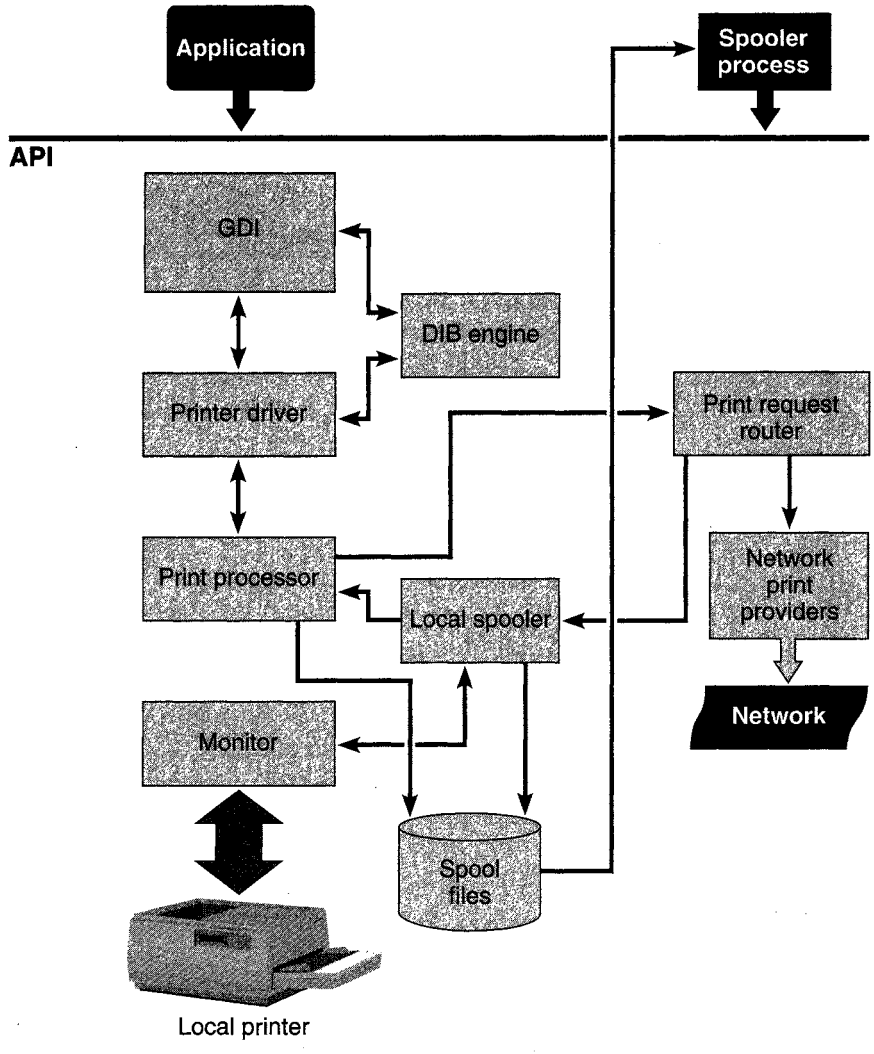


Figure 6-11.
Components of the Windows 95 printing architecture.

for subsequently handing the print job to the print request router for actual printing.

If the print job is for a local printer, the local spooler hands the data to a *print processor* that converts the journal records to a printer-specific format. Ultimately, the data stream goes to a *monitor*, and it's the monitor that actually controls the physical printer. Although it might

seem that the monitor is yet one more level of indirection in this process, it enables much more intelligent handling of a printer device. The monitor handles all bi-directional communication with the printer so that conditions such as paper out can be reported to the local spooler. This allows the user to see a useful error message, such as *paper out* or *cover open*, rather than the generic *printer not responding*. The monitor also implements the Plug and Play support for printers, enabling automatic identification of the printer, for example. The monitor design also provides a general interface that allows devices such as direct network-attached printers to function properly. As far as the spooler is concerned, the monitor is dealing with a directly connected printer. If the monitor chooses to talk NetBIOS commands to a laser printer plugged in down the hallway, so be it—the spooler doesn't care.

Rather than a print job, the application can choose to directly produce a metafile by requesting a DC using the *CreateEnhMetaFile()* API. GDI generates a metafile on disk that describes a *reference device*—a basis for the metafile contents—and a series of metafile records. Metafiles remain device independent, and an application can replay their content and direct the output to a specific device at some later time.

Microsoft plans to use enhanced metafiles as the basis for the contents of the print job data file, so all print processors will convert metafile records to device-dependent data during the despooling operation. Windows 95 will implement this for only locally attached printers, but in the future metafiles will be used for network printing. Apart from the fact that much less data gets sent across the network, the printing subsystem on the local machine is a lot simpler. It doesn't need to know much about the target printer, and the printer driver and print processor need only exist on the target machine.

Using the Universal Printer Driver

Windows 3.1 introduced a major enhancement into the printing subsystem—the universal printer driver. Like the DIB engine–mini-driver combination for display drivers in Windows 95, the universal printer driver recognizes the fact that most printers work pretty much the same way. Thus, the universal driver can encapsulate much of the printing workload, leaving the printer manufacturer free to concentrate on developing a much simpler printer mini-driver to handle the hardware-dependent interactions. Windows 95 shares the design of the printer mini-drivers with Windows NT, and a particular mini-driver will work on either system.

The universal printer driver approach has been extremely successful, and Microsoft predicts that support for over 700 different printers will be included with Windows 95 when it ships. The driver has been enhanced for Windows 95 in a few small ways, including support for 600-dpi devices and the ability to download TrueType fonts to the target printer. The mini-driver design is largely unchanged, and the philosophy remains to offload the majority of print output processing to the universal driver with the mini-driver providing only device-dependent functionality.

The world of printing is a highly complex one, and the quality of font reproduction is one of the most carefully scrutinized aspects. Adobe Systems has built a very successful business by evangelizing both its fonts and its PostScript printing technology. For many years Adobe fonts and PostScript output devices have set the standard for computer-based printing and publishing. The majority of printers deal in data streams interspersed with printer commands (the basis for the universal printer driver design), but PostScript is a page description language. The PostScript printer driver generates the description of the page to be printed with very little knowledge of the actual output device.[35] This device independence has allowed PostScript to span the range of printing devices, from $500 laser printers to high-end color film production systems costing tens of thousands of dollars. A PostScript interpreter, which resides on the output device, translates the PostScript data stream into actual hardware operations that place dots on paper or film. The universal printer driver model doesn't suit the needs of PostScript, so no use is made of the mini-driver architecture for PostScript printers.

By far the most popular laser printers for Windows systems are those in the Hewlett-Packard LaserJet series. Microsoft and Hewlett-Packard have collaborated closely on Windows printing design for several years, including the design of the TrueType font subsystem. Hewlett-Packard also has its own printer language—PCL—that is common to all the LaserJet models. Many printers feature "LaserJet emulation"—essentially meaning PCL emulation. PCL is closer to the model of the world implemented by the Windows universal printer driver, so this class of printer can use the mini-driver architecture.

35. Should you be so inclined, you can actually read the PostScript driver's output by directing it into a file.

Conclusion

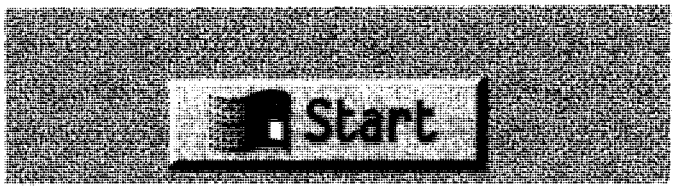
Windows 95 finally makes 32-bit Windows programming a mainstream activity. In addition to improved ease of development and compatibility with Windows NT, Windows 95 adds a number of new features to Windows. Some, such as the color matching capability, are long-awaited responses to features previously available only in competing operating systems. Other features, such as OLE and RPC, have existed before Windows 95 but never as standard components of an operating system that will be used on millions of PCs. Once again, we can all look forward to the amazing inventiveness of the software industry as it harnesses these features in new application products.

Between the API layer and the device drivers that translate application requests into operations on the bare metal, Windows 95 includes several radically new or revised subsystems. The rest of this book isolates some of these subsystems and examines them in detail. The next chapter looks at one component that everyone uses: the filesystem.

References

Microsoft Corporation. Windows 95 SDK documentation. Redmond, Wash.: Microsoft, 1994. No doubt Microsoft's product documentation will be augmented by dozens of new or warmed-over books that deal with the details of the Windows API and Windows device drivers, but I haven't seen any yet. If you program for Windows and you don't yet have a CD ROM drive—invest now. The online help files in the SDK and the CD distributed as part of the Microsoft Developer Network product are about the only sane way to approach this volume of information.

Brockschmidt, Kraig. *Inside OLE 2.0.* Redmond, Wash.: Microsoft Press, 1994. This is an intimidating book, nearly 900 pages in length. It is, however, the single most comprehensive treatment of OLE available. If OLE development is in your future, this is a book you have to tackle.



CHAPTER SEVEN

THE FILESYSTEM

Although the 32-bit API and the shell are likely to attract the highest initial interest from programmers and users, the new filesystem architecture of Microsoft Windows 95 is the base operating system component that has the most widespread impact on the system. Windows 95 continues to use the MS-DOS FAT filesystem as its default on-disk structure, but the code implementing the filesystem organization is completely new. In Windows 95, the FAT filesystem code—referred to as VFAT—is merely one piece of an entirely fresh design. These new features supported by the Windows 95 filesystem architecture affect both end users and application developers:

- Support for long filenames finally addresses the number one user complaint about earlier versions of MS-DOS and Windows. The new API support for long filenames requires developers to modify their applications, but there is an immediate and significant payback for the effort invested.
- Network support relies on the new installable filesystem architecture to allow the concurrent use of different network systems. Support for multiple network connections means that users can simultaneously access different networks without suffering through a complex setup and configuration procedure. Network software providers can develop Windows 95 network support using an interface designed to allow the integration of multiple high-performance connections.
- Users will see improved performance resulting from the implementation of the standard FAT filesystem as multithreaded 32-bit protected mode software.

- Developers specializing in the support of new hardware devices will realize the benefit of the layered filesystem design as the effort required to implement new disk device drivers is significantly reduced.

These features reflect the goals of the filesystem effort—add long filename support, improve performance, and dispense with the poorly suited MS-DOS INT 21H mechanism in favor of a properly architected interface that supports multiple filesystems. The reliance on MS-DOS has been the major weakness in every release of Windows through version 3.1. Apart from significant user frustration with the limited filenaming capabilities, there have been a number of system-level problems stemming from continued reliance on MS-DOS:

- MS-DOS[1] contains a lengthy critical section that prevents efficient multitasking of applications—particularly during heavy disk access. Retaining such a bottleneck is simply not acceptable in an operating system intended to support multithreaded applications.
- Every access to the filesystem from a Windows-based application requires the System VM to switch between protected mode and virtual 8086 mode in order to execute MS-DOS code. This is another performance hit.
- MS-DOS network support requires the network software to hook the INT 21H software interrupt and reroute the appropriate filesystem requests across the network. Every other disk-related TSR program uses the same basic interrupt hooking technique. The interface was never designed for overloading this heavily. In the case of only one network connection, this technique tends to destabilize the system, and trying to support multiple network connections is yet more problematic.
- Proprietary solutions have led to a profusion of filesystem interfaces designed to support CD ROM devices, SCSI adapters, tape devices, and other devices. Even when a particular

1. Note that references to MS-DOS in this chapter mean MS-DOS releases up to and including version 6.22. If there is an MS-DOS version 7.0, it will incorporate the same filesystem architecture as Windows 95.

interface proves to be popular under MS-DOS, supporting the interface in Windows is by no means a straightforward task.

Elements of the new filesystem have been under development since early 1991, and much of the new filesystem design appeared for the first time with the November 1993 release of Microsoft Windows for Workgroups version 3.11. This release of Windows included the protected mode implementation of the MS-DOS FAT filesystem and support for multiple network connections. However, the Windows for Workgroups release did not include either the long filename capability or the full features of the base OS to be introduced with Windows 95.

In this chapter, we'll examine the features that enable the coexistence of multiple filesystems and the details of the support for what Windows 95 calls *block devices*[2]—principally disk and tape drives that are local to the host system. Network support relies on the new filesystem architecture also, with Windows 95 classifying the higher layer of any network connection software (usually called the *redirector*) as a network filesystem. In Chapter Nine, on Windows 95 networking, we'll revisit this particular filesystem type in more detail.

Overview of the Architecture

There are many individual components of the new filesystem architecture. In fact, to refer to it as "the filesystem" is to be rather inaccurate. The design relies on a layered approach that places the *installable filesystem manager* (IFS) at the highest level and a collection of *port drivers*, or *miniport drivers*, at the lowest level, where they interface to individual hardware devices. Within the boundaries set by these components, the system can support several different active filesystems. Windows 95 supports some—such as the FAT filesystem—directly. Support for non-Microsoft filesystems comes from installable modules supplied by other vendors. If you're familiar with the disk subsystem design of Windows NT, you'll notice a lot of similarities to it in the Windows 95 design. Figure 7-1 on the next page illustrates the principal components of the filesystem architecture.

2. Microsoft referred to the complete block device driver subsystem as "Dragon" during development. This subsystem deals only with local block devices and not with network support.

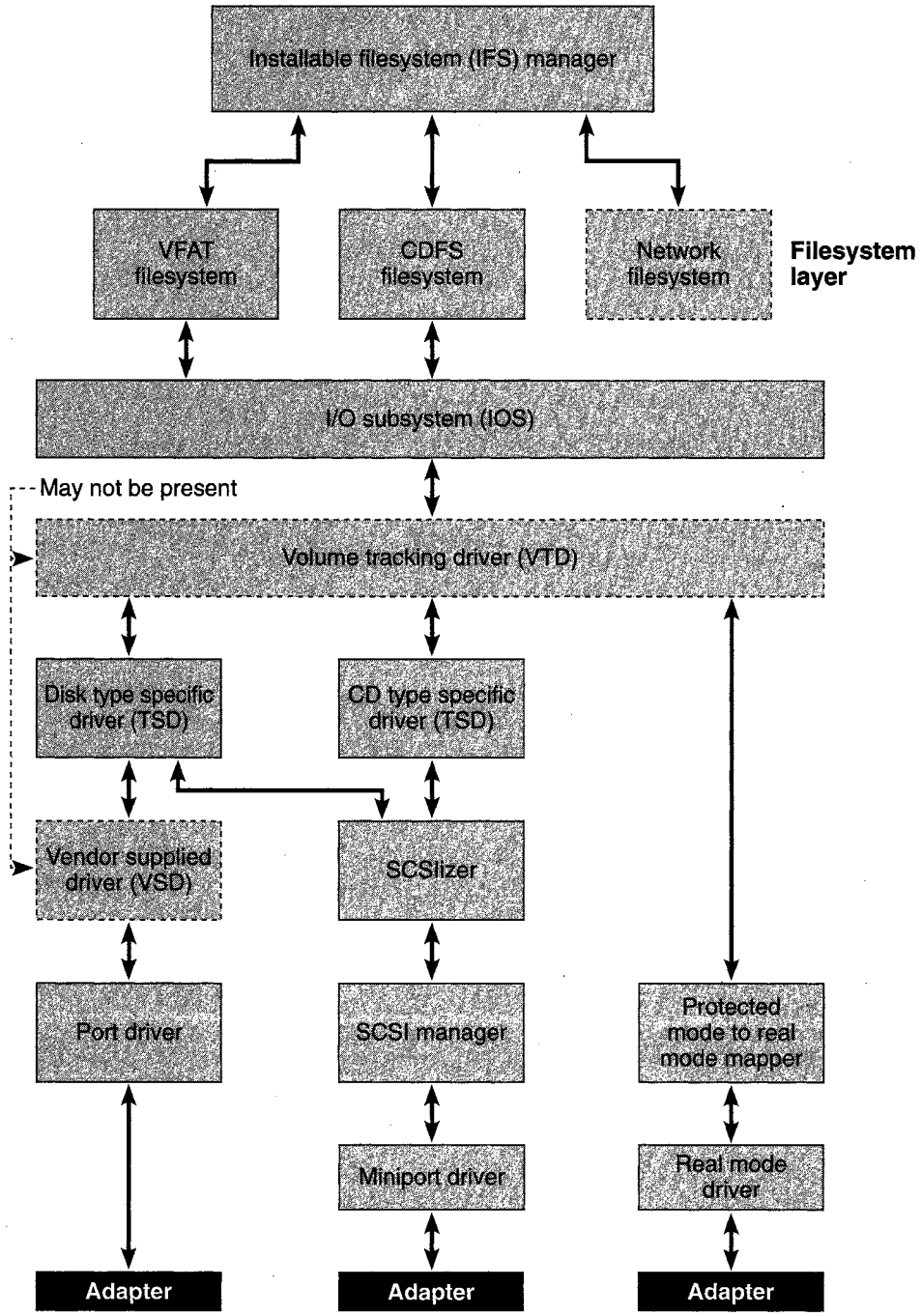


Figure 7-1.
Windows 95 filesystem architecture layers.

The choice of a layered design controlled by the IFS aims to resolve the problems inherent in using the MS-DOS INT 21H interrupt as the solitary interface to every filesystem function. Network systems and other popular products such as caching software and disk compression TSRs all hook INT 21H to inspect every file request for possible rerouting. Since there's no well-defined order for these TSRs, or any published interface between them, the interactions can cause problems. And conflicts among different vendors' products usually highlight any review of an MS-DOS release. Even when various products can be made to work well together, the user might have had to indulge in hand to hand combat with the CONFIG.SYS and AUTOEXEC.BAT files first. The Windows 95 filesystem design fixes this situation by providing many levels in which add-on components can be installed. Each layer has defined interfaces with the layers above and below, which enables each component to collaborate smoothly with its neighbors. The new filesystem architecture relies on the dynamic VxD loading capability of Windows 95 to load many of its lower-level components.

Figure 7-1 illustrates only a small number of the possible layers in the filesystem—although these are the components you'd expect to find in a "standard" system. The filesystem design supports as many as 32 layers from the *I/O subsystem* (IOS) down. Layer 0 is the layer adjacent to IOS, and layer 31 is closest to the hardware. On initialization, a component registers itself with IOS and declares the layers at which it wishes to operate. To operate at more than one level, a module has to supply IOS with different entry points—one per required level. Above IOS are the filesystems themselves and the *installable filesystem manager* (IFS manager). Let's take a brief look, from the top down, at the functions of the common layers and at the components you'd expect to find in them:

The **IFS manager,** at the highest layer, is a single VxD that provides the interface between application requests and the specific filesystem addressed by an application function. The IFS manager accepts both dynamically linked API calls from Win32 applications and INT 21H calls generated by Win16 or MS-DOS applications. The IFS manager transforms the API requests into calls to the next layer, the filesystem layer.

The **VFAT,** in the filesystem layer, is the protected mode implementation of the FAT filesystem. VFAT is an example of a *filesystem driver,* or *FSD.* Each FSD implements a particular filesystem

organization. An FSD executes requests made by the IFS manager on behalf of an application. The IFS manager is the only module that calls an FSD; applications never call an FSD directly. VFAT itself is a 32-bit module written as reentrant code, allowing multiple concurrent threads to execute filesystem code.

The **CDFS,** in the filesystem layer, is the protected mode implementation of an ISO 9660–compliant CD ROM filesystem. It's another example of an FSD. Again, it's 32-bit protected mode, reentrant code. In most cases, CDFS will replace the real mode MSCDEX TSR that's currently used to support CD ROM devices, so there'll also be a protected mode execution path all the way to the CD ROM hardware.

The **I/O subsystem,** or IOS, is the highest layer of the block device subsystem. The IOS component is permanently resident in memory and provides a variety of services to the other filesystem components, including request routing and time-out notification services.

The **volume tracking driver,** or VTD, in the layer below the IOS layer, is the component responsible for managing removable devices. Typically, such a device is a floppy disk, but any device that conforms to what Windows 95 calls "the removability rules" can use the VTD services. The most important job of the VTD is to make sure that the correct disk or device is in the drive. If you exchange a floppy disk while an application still has a file open, it's the VTD that initiates a complaint.

A **type specific driver,** or TSD, in the layer below the VTD layer, manages all devices of a particular type—for example, hard disks or tape devices. A TSD validates requests for the device type that it controls and carries out the logical to physical conversion of input parameters. Note that a TSD relates more to devices of a specific logical type—for example, compressed volumes—than to devices of a specific hardware type.

A **vendor supplied driver,** or VSD, is the layer in which another vendor can supply software that intercepts every I/O request for a particular block device. At this level, for example, you could modify the behavior of an existing block device driver without

having to supply a completely new driver. A data encryption module is one example of a potential VSD.

A **port driver,** or PD, is a component that controls a specific adapter. On an ISA bus personal computer, for example, there would probably be an IDE port driver. A port driver manages the lowest levels of device interaction, including adapter initialization and device interrupts.

The **SCSIizer** translates I/O requests into SCSI format command blocks. Usually these will be one SCSIizer module for each SCSI device type—CD ROM, for example.

The **SCSI manager** is a component that allows the use of Windows NT miniport drivers in Windows 95. Literally, you can use the same binaries for both Windows NT and Windows 95. The SCSI Manager provides a translation between the Windows NT miniport driver and the upper layers of the filesystem.

A **miniport driver** is specific to a SCSI device. In conjunction with the SCSI manager, it carries out the same function as a port driver, but for a SCSI adapter. Miniport drivers for Windows 95 share the design and implementation rules for Windows NT miniport drivers.

The **protected mode mapper** is a module that enables the use of existing MS-DOS drivers under Windows 95. For compatibility, it's essential to allow existing drivers to run under Windows 95. The protected mode mapper disguises real mode drivers for the benefit of the new filesystem modules—so that they don't have to take account of the different interface.

A **real mode driver** is an existing MS-DOS–style device driver that must run in virtual 8086 mode.

Long Filename Support

The widespread ramifications of the new long filename support in Windows 95 guarantee that every user and programmer will have to pay attention to the feature. Microsoft has encouraged (actually exhorted) Windows application developers to incorporate support for this feature as soon as possible. Microsoft's providing long filename support for

MS-DOS applications underscores this level of encouragement—if you have a product that is available in both Windows and MS-DOS versions, there's no barrier to upgrading both versions.

For users, long filenames are a real benefit. The need to learn rules for filenaming essentially disappears, together with the frustrating inadequacy of the current MS-DOS 8.3 convention. Unfortunately, it's impossible to simply throw a switch and have every application and every existing disk in the world suddenly support long names. For some period of time, applications that support only the old filename conventions will live alongside those that offer access to the new naming scheme. In Figure 7-2, you can see again Chapter Five's example of the support that Windows 95 has to provide to applications in order to allow the parallel existence of short and long filenaming. In the first screen, a file created with a long name is visible in both the Windows 95 shell and the Windows 95 version of COMMAND.COM. The second screen shows the Open dialog for a Windows 3.1 application running under Windows 95. The Windows 3.1 application doesn't handle long filenames, so the system has to generate an equivalent short name that allows the unmodified application to access the file.

This creation of short name equivalents is a fundamental feature of the new filesystem architecture. It would be nice to assume that it's going to be a short-lived feature, but it's probably around to stay. A short filename is not simply a truncated or mutated version of the long name—several rules govern both the format of the name and its behavior in response to different filesystem operations. We'll look at those details later in this chapter. First we'll look at the disk structure for storing the new long filename format.[3]

Storing Long Filenames

The compatibility requirements Windows 95 has to meet meant that it was impossible to simply change the existing FAT filesystem disk format. Although most applications deal with the disk by means of the defined operating system interfaces, there are many popular utility programs that directly inspect and modify the disk format. Virus scanning programs, disk repair utilities, optimizers, and many other programs depend on the on-disk structure of the FAT filesystem.

3. Late in the project Microsoft began to refer to the long filename as the "primary file name" and to the short name as the "alias" or "alternate name." For clarity's sake, I'll continue to use "long" and "short" in this chapter.

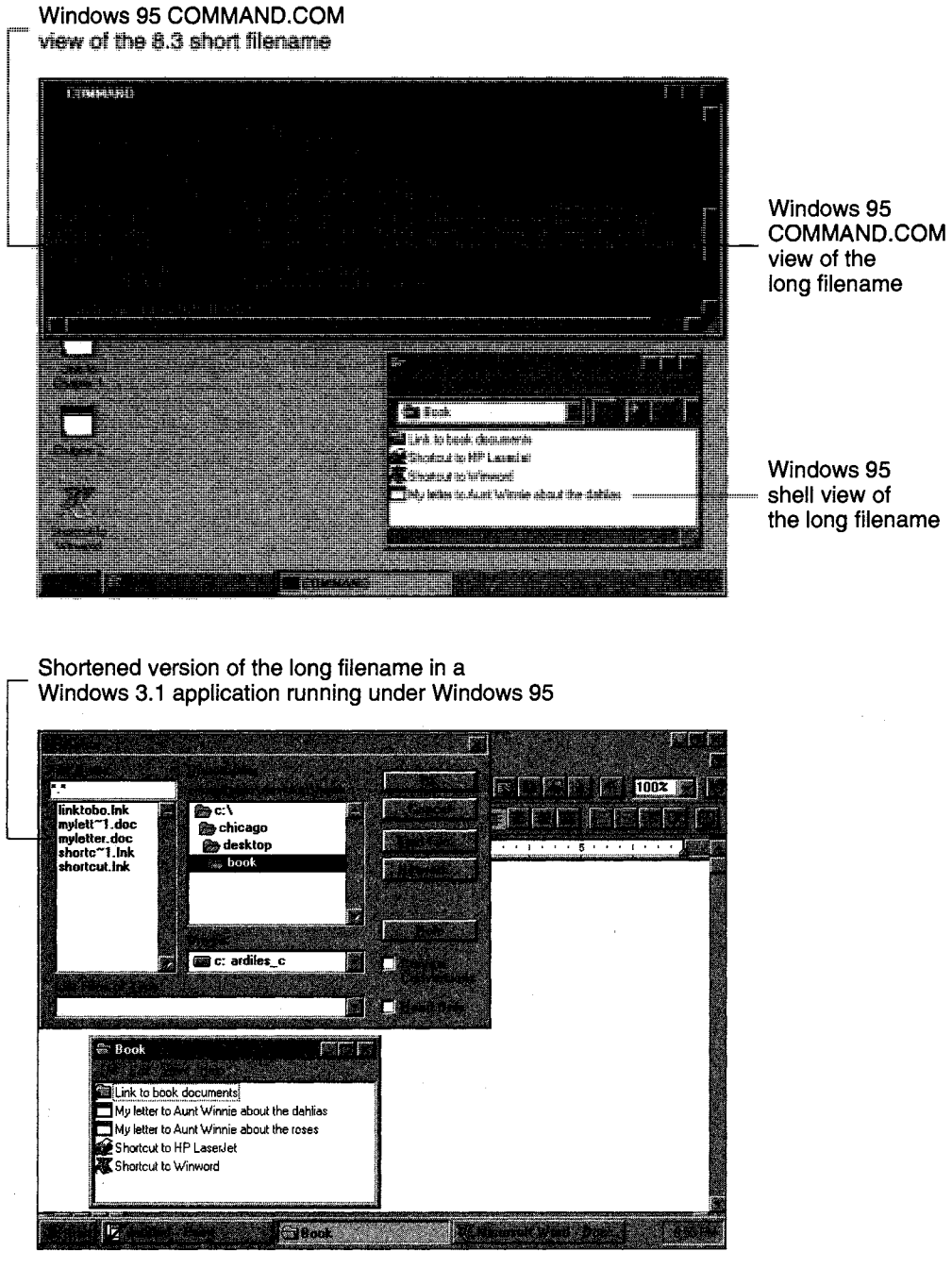


Figure 7-2.
A long filename and the short version.

Modifications to that structure would have caused all of these programs to fail. In some cases, the failure could well have resulted in loss of the user's data—a risk that was obviously unacceptable. The technique for implementing long filename support relies on a little design trickery and a great deal of careful implementation and compatibility testing.[4]

Figure 7-3 shows the format of a FAT filesystem directory entry for a short name (that is, for a filename conforming to the existing 8.3 naming conventions). The new VFAT filesystem supports both long and short names and, apart from its not using the "last date accessed" field, the 32-byte short name directory entry is identical in format to the format supported by previous versions of MS-DOS. Short names in both the FAT and VFAT filesystems have the following rules associated with them:

- The name can consist of as many as eight characters with an optional three character extension.
- Valid characters in the name are letters, digits, the space character, any character with a character value greater than 7FH, and any of the following:

$	dollar sign
%	percent symbol
' and '	open and end single quotation marks
'	foot mark (apostrophe)
-	hyphen
_	underscore
@	at sign
~	tilde
`	grave accent
!	exclamation mark
(and)	left and right parentheses
{ and }	left and right braces
#	pound sign
&	ampersand

4. The implementation trick prompted Microsoft to pursue a patent application for the underlying technique. Pursuit of the patent was abandoned, however.

- The full path for a file with a short name can be as many as 67 characters, not including a trailing null character.
- The FAT and VFAT filesystems always convert shortened names that include lowercase letters to uppercase only. This avoids potential problems with matching filenames. For example, the filename Afile.txt is converted to AFILE.TXT and will match the strings afile.txt, afile.TXT, AFILE.txt, and any other possible combination of uppercase and lowercase letters.

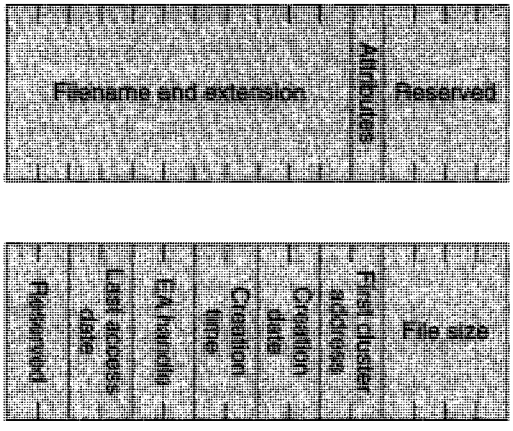


Figure 7-3.
Short name directory entry format for the FAT filesystem.

The implementation technique for long filenames relies on the use of the short name directory entry attribute byte. Setting the least significant 4 bits of this byte (that is, the value *0FH*) gives the directory entry the attributes *read only, hidden, system file,* and *volume.* Adding the *volume* attribute produces an "impossible" combination. Amazingly, Microsoft's testing showed that this combination didn't disturb any existing disk utilities. Unlike other invalid combinations, which cause disk utilities to try to "fix" the problem and thus destroy the data, the *0FH* attribute value protects the directory entry from modification.

Despite the encouraging test results, Microsoft knew there was a possibility that some untested disk utility could destroy data. To avoid such a potential catastrophe, the team came up with an "exclusive volume lock" API that an application must call before Windows 95 will allow direct disk writes (MS-DOS INT 13H and INT 26H).

The "exclusive volume lock" API is accessible either as a new MS-DOS interrupt (INT 21, function 440D, major code 08) or by means of the Win32 *DeviceIoControl()* API. If an application has not been granted exclusive volume access before it tries a direct disk write, the attempted write operation will fail.

To avoid forcing users to get updates to their existing disk utilities, Microsoft planned to include a command-level interface to allow a user to run an older disk utility within a "wrapper" function that obtained and released the volume lock on behalf of the application.

Windows 95 uses multiple consecutive short name entries for a single long name—protecting each of the 32-byte entries by using the *0FH* attribute. The rules for long filenames are different from those for short names:

- Every long name must have a short name associated with it. The file is accessible by means of either name.
- A long filename can contain as many as 255 characters, not including a trailing null character.
- Valid filename characters include all the characters usable in short names plus any of the following:

+	plus sign
,	comma
;	semicolon
=	equals sign
[and]	left and right square brackets

- Leading and trailing space characters within a name are ignored.
- The full path for a file with a long name can be as many as 260 characters, not including a trailing null character.
- The system preserves lowercase characters used in long filenames.

Within a single directory cluster, a long filename directory entry is laid out according to the format shown in Figure 7-4. A long filename component cannot exist without the associated short name entry. If it does, that's an indication that the disk is corrupt.

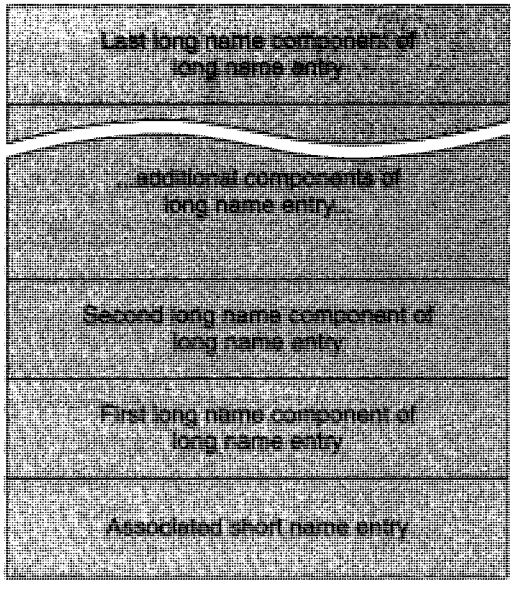


Figure 7-4.
Directory cluster format for a long filename.

Each 32-byte component of the long name entry contains a *sequence* number, the protective *attribute* byte, a *type* value, and a *checksum.* The *sequence* number helps Windows 95 recognize any inconsistent modifications to the directory structure. The *type* field identifies the component as either LONG_NAME_COMP (a component of the long name) or LONG_CLASS (a 32-byte entry that contains class information for the file). If the component is part of the name, most of the 32-byte entry is used to store filename characters. If it's the single class component for that file, the entry holds the class information. Notice that the system stores long filenames using the Unicode character set—meaning that each filename character requires 16 bits.[5] The *checksum* field in each component entry is formed from the short name associated with the file. If the short name is ever changed outside the Windows 95 environment (for example, the file is renamed on a floppy disk using MS-DOS version 5.0), Windows 95 can recognize the long name components as no longer valid. Figures 7-5 and 7-6 on the next page show the name and class component formats for these entries.

5. Unlike Windows NT, Windows 95 did not switch entirely to using the Unicode character set for its internal representation. This is one instance in which the change was made.

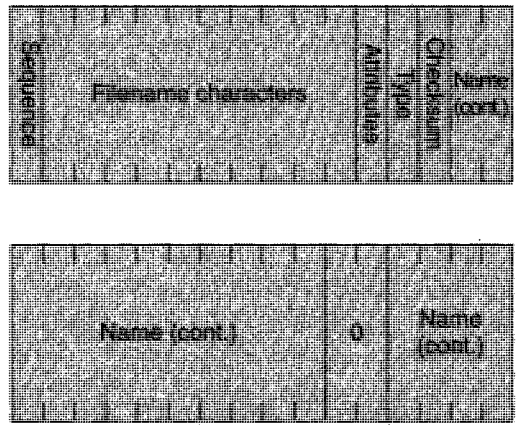


Figure 7-5.
Long filename directory entry format.

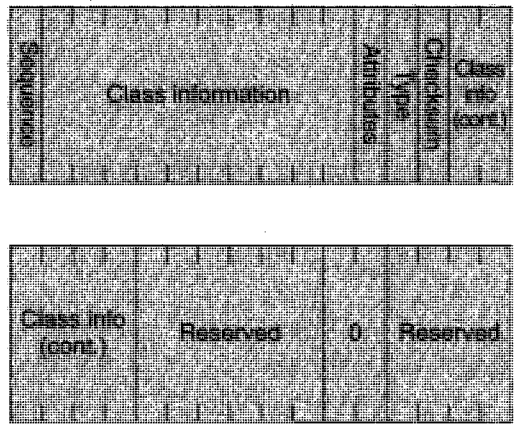


Figure 7-6.
Long filename class information directory entry format.

Generating Short Filenames

A whole series of rules defines how to generate a short filename to associate with a long filename—and we're not going to examine every last nuance of the algorithms. The principal problem is to generate a unique short filename that doesn't conflict with an existing short name. Similarly, if an older application creates a new file with a short name, that name can't clash with an existing short name associated with a long filename. Fortunately, these issues aren't really visible to

application programs—employees of companies that produce disk utilities are the only people who will have to delve into the intricacies of the naming system.[6] Here's a summary of most of the important rules used in filename creation:

- Creating a file by using a short name API (that is, by means of the older INT 21H interface) results in a long name that's identical to its associated short name. If a matching long name already exists, the create operation fails—the same behavior you'd see if you tried to create a file with a nonunique short name.
- Creating a file by using a long name API always results in the creation of the associated short name at the same time.
- If the long name is to be a valid short name, it must be unique. For example, if a short name AFILE.TXT already exists, an attempt to create the long name AFile.Txt will fail—this test is always case insensitive.
- If the long name is not a valid short name, the system carries out a series of name truncation and translation operations in an attempt to arrive at a valid short name. Note to Kathleen (Review Comments).Document, for example, would successively translate to[7]

  NotetoKathleen_ReviewComments_.Document
  NOTETOKA.DOC
  NOTETO~1.DOC

- The system would then modify the ~1 suffix to ~2, ~3, and so on until it came up with a unique short name. If a ~9 suffix didn't work, NOTETO~9.DOC would become NOTET~10.DOC, NOTET~11.DOC, and so on.

MS-DOS Support for Long Filenames

To help promote the use of long filenames across all application types, Windows 95 extends the MS-DOS INT 21H interface to allow the use of long names. This extension involves adding new functions that are

6. This assumption begs the question of whether application developers will invent schemes to assist users in the translation between long and short names.

7. This is not a description of how the algorithm actually proceeds; it simply serves to illustrate the steps involved.

directly equivalent to Win32 API functions and modifying existing MS-DOS functions that deal with filenames. The calls to the new and modified INT 21H functions continue to use the standard MS-DOS calling conventions with parameters passed and returned in registers. And the functions are still 16-bit code; the fact that the functions are equivalent to Win32 APIs doesn't change the memory mode. Here's a summary of the new functions—all numbers are hexadecimal values:

MS-DOS Function	Equivalent Win32 Function
INT 21 function 4302	*GetVolumeInformation()*
INT 21 function 57	*GetFileTime(), SetFileTime()*
INT 21 function 6C	*CreateFile(), OpenFile()*
INT 21 function 7139	*CreateDirectory()*
INT 21 function 713A	*RemoveDirectory()*
INT 21 function 713B	*SetCurrentDirectory()*
INT 21 function 7141	*DeleteFile()*
INT 21 function 7143	*GetFileAttributes(), SetFileAttributes()*
INT 21 function 7147	*GetCurrentDirectory()*
INT 21 function 714E	*FindFirstFile()*
INT 21 function 714F	*FindNextFile()*
INT 21 function 7156	*MoveFile()*
INT 21 function 716C	*CreateFile(), OpenFile()*
INT 21 function 72	*FindClose()*

Notice that in most cases the functions use new function codes—the other parameters are identical. The new function codes are necessary because the system needs to know whether the application is dealing with short names only or with the extended namespace. For example, an application using INT 21H function 41H to delete a file could pass the filename ABIGBADNAME.TXT as the filename parameter. The filename is illegal under the "old" semantics, although it is a perfectly valid long name. If the INT 21H function 41H call were simply overloaded to allow the use of long names, this semantic error would go undetected. Thus, the new INT 21H function 7141H is the only way to delete a file with a long name, and the same rules apply to the other new name-related functions.

Long Filenames on Other Systems

A file's short name is used by applications that haven't been modified to handle long names, but reading long filenames isn't just an application issue. Long names also disappear on other, otherwise compatible, systems such as Windows NT (versions 3.0 and 3.1), OS/2, and earlier versions of MS-DOS (versions 6.22 and earlier). In most cases, the operating system can't handle the new form of directory entry. In the case of OS/2, the implementation of long filenames is different and incompatible.[8]

The restriction also applies to Windows 95 when the user is in single MS-DOS application mode: the long names are invisible. Access to any file can always be accomplished by using the short name, however—regardless of the host operating system.

Installable Filesystem Manager

The IFS manager in Windows 95 provides features similar to those in other implementations of this type of filesystem design. The development team actually looked very hard at the Windows NT IFS implementation to see whether the code could be adapted for use in Windows 95, but the internal differences between the two operating systems meant that a new implementation was required for Windows 95. Where it made sense to, though, the Windows 95 team used the design of the Windows NT IFS, and they retained the same names for entry points and the like.

The basic role of the IFS manager is to accept all filesystem API calls, convert each to the appropriate IFS interface call, and then pass the request to the target filesystem driver. The target FSD is responsible for interpreting the function call according to its private semantics; the IFS manager simply gets the information to the FSD. The IFS manager is the common target for both Win32 API calls and MS-DOS INT 21H filesystem functions. Once the IFS manager is in control, the execution path for the filesystem call remains a 32-bit protected mode path all the way to the hardware and back, with two possible exceptions.

8. Windows NT does support long filenames within the NTFS filesystem, but versions 3.5 and earlier don't support long filenames within a FAT filesystem. The Windows NT and Windows 95 long name schemes are not compatible.

- The filesystem code has to use a real mode device driver to interface with the hardware.
- The filesystem code has to call a real mode TSR that has hooked the MS-DOS INT 21H interrupt.

In either case, the filesystem code calls the real mode component (using virtual 8086 mode) within the context of the VM initiating the filesystem request.

The IFS manager loads during system initialization. It is always in memory, and it must be present before any individual FSD can load. The IFS manager allows several FSDs to execute concurrently.[9] Each FSD registers itself with the IFS manager during its own initialization, passing the IFS manager a table of entry points that will be used in subsequent filesystem calls. Once active, the IFS manager chooses which FSD to call to resolve a particular filesystem request in one of three ways:

- If the API provides a path as a parameter, the IFS manager uses either the embedded drive letter or the whole name to determine the target FSD. For example, a file open call specifying C:\AUTOEXEC.BAT will be passed to the local VFAT FSD.
- If the API passes a file handle obtained, for example, as the result of a previous file open call, the IFS manager uses the handle as an index into a *system file handle* structure. The entry in this structure identifies the target FSD and the FSD-specific handle for the IFS manager to use when it routes the request to the FSD.
- In the event that the IFS manager can't identify the target FSD, it will call each FSD in turn until one of them agrees to accept the request. When the user inserts a new floppy disk, for example, the IFS manager calls each FSD, asking it to *mount* the new volume. To mount the volume, the FSD must recognize the media format; if it doesn't, the IFS manager passes the mount request to the next FSD.

9. The initial design allowed as many as 10 local filesystem drivers and 10 remote filesystem drivers to execute at the same time.

Calling a Filesystem Driver

The interface between the IFS manager and an FSD relies on the use of a single data structure called an IOREQ. This structure is a large data object (approximately 100 bytes) containing many individual fields—only some of which are used in each call between the IFS manager and an FSD. Each call to the filesystem code from an application causes the IFS manager to fill in an IOREQ structure and pass it to the target FSD. For performance reasons, the IFS manager passes a pointer to an IOREQ structure rather than the entire data object. The FSD directly modifies fields in the IOREQ structure to return results to the IFS manager. Before returning to the application, the IFS manager examines the IOREQ structure and extracts both the information that it retains internally and the relevant return parameters for the application. Figure 7-7 shows the format of the IOREQ structure.

```
struct ioreq {
    unsigned int    ir_length;  // length of user buffer (eCX)
    unsigned char   ir_flags;   // misc. status flags (AL)
    uid_t           ir_user;    // user ID for this request
    sfn_t           ir_sfn;     // System File Number of file
                                //  handle
    pid_t           ir_pid;     // process ID of requesting task
    path_t          ir_ppath;   // unicode pathname
    aux_t           ir_aux1;    // secondary user data buffer
                                //  (CurDTA)
    ubuffer_t       ir_data;    // ptr to user data buffer
                                //  (DS:eDX)
    unsigned short  ir_options; // request handling options
    short           ir_error;   // error code (0 if OK)
    rh_t            ir_rh;      // resource handle
    fh_t            ir_fh;      // file (or find) handle
    pos_t           ir_pos;     // file position for request
    aux_t           ir_aux2;    // misc. extra API parameters
    aux_t           ir_aux3;    // misc. extra API parameters
    pevent          ir_pev;     // ptr to IFSMgr event for async
                                //  requests
    fsdwork_t       ir_fsd;     // Provider work space
};  // ioreq
```

Figure 7-7.
The IOREQ data structure.

To ease the implementation burden for developers, Microsoft used C language calling conventions to define the interface between the IFS manager and an FSD. So, if you want to get into the business of developing new filesystems for Windows 95, at least you don't have to write them in assembly language. The IFS manager also provides a set of services, callable by FSDs, that fulfill common requirements such as heap memory management, debugging, event signaling, and filename string manipulation.

If the IFS manager is to recognize and use an FSD, the FSD must first register itself using an IFS manager service. The two principal services are *IFSMgr_RegisterMount()* and *IFSMgr_RegisterNet()*, which announce, respectively, the presence of an FSD capable of managing local filesystems or one devoted to the management of a network resource. No meaningful interaction can occur between the IFS manager and an FSD until the FSD has declared its presence using one of the IFS registration services. In each call, the FSD passes a single entry point address to the IFS manager. The entry point address identifies the function called by the IFS manager the first time the manager calls out to the FSD.

Filesystem Drivers

Each Windows 95 FSD is a single VxD responsible for implementing the particular semantics of its native filesystem. Knowledge of a particular filesystem layout exists entirely within the code of an FSD. The IFS manager deals only in handles, and the lower layers of the filesystem deal mostly in byte offsets and counts. Only the FSD knows how to get from an application-supplied name to particular data on a filesystem volume. FSDs can control either local or remote filesystems. Depending on how the FSD registers itself with the IFS manager (local or remote), the FSD must provide a number of individual entry points for use by the IFS manager. Not every FSD must support every function defined as part of the IFS interface—the mandatory entry points depend largely on whether the filesystem type is local or remote. In addition to the two major filesystem types, Windows 95 recognizes a *mailslot* filesystem type that can be used to provide inter-application messaging services.

The single entry point provided by the FSD when it registers with the IFS manager identifies either the *FS_MountVolume()* function (for local filesystems) or the *FS_ConnectNetResource()* function (for remote

filesystems). These functions are among the set of standard entry points defined for the IFS manager interface. When the IFS manager calls the single entry point, the FSD will return a pointer to a table of additional entry points. Subsequent calls from the IFS manager to an FSD go directly to the specific function using one of these new entry points. A called function may return yet more entry point addresses. It's all like peeling away the layers of an onion. The FSD returns these function pointers to the IFS manager on what you can think of as an as needed basis, and gradually the IFS manager learns how to call every entry point in a particular FSD. (Until a file is open, for example, the FSD won't provide the IFS manager with a way to call either the file positioning function or the file locking function.)

The IFS manager calls the initial *FS_MountVolume()* entry point for local filesystems as the result of either the first access to a device or a change to the media. The call asks the FSD to try to mount the volume (the VOL_MOUNT operation). It's up to the FSD to determine whether it recognizes the device media format. If it does, it returns a volume handle and a pointer to the initial table of functions to the IFS manager. The handle is used to identify the volume in subsequent calls to the FSD. For disks, the volume handle will identify either a hard disk partition or a specific floppy disk. The IFS manager initiates the removal of all access to a volume by calling the *FS_MountVolume()* entry point, specifying an unmount (the VOL_UNMOUNT operation).

For network filesystems, the IFS manager calls the function *FS_ConnectNetResource()* with a network path for the target resource. As with local filesystem access, the FSD must determine whether it should be responsible for managing the particular resource. If it is, it returns a handle and a function table to the IFS manager. If it isn't, the FSD returns an error and the IFS manager must carry on, looking for the correct FSD to match to the network resource.

FSD Entry Points

The next page contains a summary of all the defined entry points for a filesystem driver.[10]

10. There's also a set of entry points used specifically to implement named pipes — Microsoft's preferred network-based, high-level inter-application communication mechanism. Local FSDs don't have to implement these services.

FSD Entry Point Name	Purpose
FS_CloseFile()	Close an open file
FS_CommitFile()	Flush any cached data for a particular file
FS_ConnectNetResource()	Call initial remote filesystem entry point
FS_DeleteFile()	Erase a named file
FS_Dir()	Call directory operations (such as create and remove)
FS_DisconnectNetResource()	Remove a network connection
FS_FileAttributes()	Set and retrieve file and filesystem information
FS_FileDateTime()	Perform date and time management on a file
FS_FileSeek()	Perform file positioning operations
FS_FindClose()	Close an *FS_FindFirstFile()*-initiated sequence
FS_FindFirstFile()	Initiate a filename search sequence
FS_FindNextFile()	Continue an *FS_FindFirstFile()* sequence
FS_FlushVolume()	Flush all cached data for the volume
FS_GetDiskInfo()	Get information about disk format and free space
FS_GetDiskParms()	Call the older MS-DOS DPB function (INT 21H function 32H)
FS_Ioctl16Drive()	Call the older MS-DOS I/O control operations (INT 21H function 44H)
FS_LockFile()	Call record-locking functions
FS_MountVolume()	Call initial entry point for local filesystems
FS_OpenFile()	Call file open and create functions
FS_ReadFile()	Call input operations
FS_RenameFile()	Call file rename operation
FS_SearchFile()	Implement MS-DOS find first and find next operations (INT 21H functions 11H, 12H, 4EH, and 4FH)
FS_WriteFile()	Call file output operations

I/O Subsystem

IOS is the Windows 95 system component responsible for loading, initializing, and managing all of the lower-level filesystem modules. (Typically, these modules are port drivers directly concerned with the

underlying hardware.) IOS also provides services to FSDs to allow them to initiate device-specific requests. IOS must be permanently resident in memory. It's loaded from the IOS.386 file early in the system initialization process.

The IOS and device driver layers rely on the use of a large number of interlinked control blocks[11] coupled with the standard VxD service interface and an implementation technique referred to as a *calldown chain.* An FSD will prepare a request for a device by initializing a control block and passing it to the *IOS_SendCommand()* service. The control block used in such a request is called an *I/O packet,* or *IOP.* IOS uses the IOP to control the passage of the device request down and back up the driver hierarchy. Most other control blocks used by IOS are hidden from the higher layers, and an FSD doesn't have to worry about the allocation or management of device-specific control blocks. We'll look at the role of several other control blocks within the filesystem architecture as we examine the components of the IOS and its lower-level driver modules.

IOS itself operates in one of two roles—as the managing entity when specific device requests are in progress, or as the provider of a number of centralized services that any device driver can call. Here are the three basic VxD services offered by IOS:

IOS_Register()	The service used by device drivers to register their presence in the system. Without the driver's prior registration, IOS can't interact with the driver.
IOS_SendCommand()	The service used to initiate specific device actions such as data transfers and disk ejection.
IOS_Requestor_Service()	The service that provides a small number of individual functions such as the functions that obtain information about a disk drive's characteristics.

In addition, a wide range of services (called *IOS service requests*) are used by drivers to control their interaction with IOS. Calling these services first requires the device driver to register itself with IOS.

11. Over 10 different data structures are defined for the I/O subsystem. Many of these data structures appear in multiple interlinked lists.

During registration, IOS provides the driver with the addresses of the entry points to call when making subsequent service requests.

Device Driver Initialization

IOS takes on the job of loading all the device drivers and requesting their initialization. IOS loads a driver in response to a request from the configuration manager (part of the Plug and Play subsystem) or because of the presence of the driver in the SYSTEM\IOSUBSYS directory. Configuration manager–initiated loading occurs when the Plug and Play subsystem detects the presence of a particular device. IOS force loads the remaining drivers in the IOSUBSYS directory. At the completion of the entire boot process, IOS will send every driver a "boot complete" message. If a loaded driver failed to recognize any hardware it can support, it can unload itself from memory at this point. There are provisions for the system to load older (non-IOS-compliant) drivers by simply including them in the SYSTEM.INI file, as Windows 3.1 does today. Drivers that conform to the new design are all dynamically loadable VxDs and must cooperate with IOS in building the layers of the device control subsystem.

Once IOS has loaded all the necessary device driver modules, the initialization process begins. The initialization of a specific driver module occurs when IOS sends to the driver module's control procedure the VxD message SYS_DYNAMIC_DEVICE_INIT. The driver must register itself with IOS by calling the *IOS_Register()* service with the address of a *driver registration packet*, or *DRP*. The DRP is a data block containing information such as the driver name and the driver's particular characteristics. One of the implementation rules for device drivers is that the address of the driver's DRP structure must appear in the VxD header for the driver module. The appearance of the address in the VxD header allows IOS to examine the DRP structure before it sends the initialization message. Three fields in a DRP are vital to the initialization process:

DRP_ilb	Contains the address of an *IOS linkage block*, or *ILB*. IOS fills the ILB structure with the addresses of several IOS entry points used in subsequent calls to IOS.
DRP_LGM	Contains the *load group mask*, or *LGM*, used during the device initialization process.

DRP_aer — Contains the address of the driver's *asynchronous event routine,* or *AER.* This asynchronous event function is called by IOS to notify the driver of any asynchronous event—for example, the completion of a time-out.

The load group mask is a 32-bit quantity defining the levels at which a driver module wants to operate. IOS sends the initialization message to the driver once for each level at which the driver module wants to register—proceeding from level 31 (the lowest) up to level 0. Since IOS can examine every driver's *DRP_LGM* field before any initialization, it's able to figure out the order in which to carry out the initialization process. IOS completes the initialization for every driver at one level before it moves upward to the next layer. So the initialization of all layer 31 drivers occurs first, followed by all layer 30 drivers, and so on. Several standard levels are defined, so almost every driver will simply use one of these level numbers as the value of its load group mask field.

IOS uses the driver's asynchronous event entry point during initialization to allow the driver to carry out private setup operations, so the driver receives control back from IOS at well-defined points during the initialization process. Among other activities, the driver creates *device data blocks* (*DDBs*) that hold control information about the device and may add itself to the device calldown chain. The driver can also specify its requirements for private workspace within an IOP during initialization. Once the initialization is complete, IOS calculates the final size of an IOP for a particular device: the size of a fixed header plus the size of an *I/O request* (*IOR*) structure, plus the sum of the sizes of all private workspace areas. Whenever an FSD subsequently requests the allocation of an IOP, the IOP size is known from this initial calculation. Also, as an individual I/O request proceeds, driver modules at different levels will have access to the necessary private workspaces at known offsets within their IOPs.

Controlling an I/O Request

As we saw earlier, the local block device subsystem deals in terms of volumes—a hard disk partition or a floppy disk, for example. For each active volume, IOS maintains a data structure called a *volume request packet,* or *VRP.* Calling IOS's *IOS_Requestor_Service()* and specifying the *IRS_GET_VRP()* function will return the address of the VRP for a particular volume. Within the VRP are the address of the entry point

within IOS that an FSD must use when it initiates I/O requests, and the size of the IOP necessary for requests to this volume.

An FSD initiates an I/O request by allocating an IOP of the correct type and size (this allocation is another IOS service), filling in the IOR structure (contained within the IOP), and passing the IOP to IOS.

IOS itself uses a structure called a *device control block,* or *DCB,* to manage much of its interaction with a particular device. A DCB is a large (256-byte) data structure that contains information about the device, such as the total number of sectors and the number of sectors per track for a disk drive. Whereas an application I/O request initially results in the creation of an IOP that provides a logical description of the request, the DCB holds information about many of the physical aspects of the device that must satisfy the application I/O request. Applications and, indeed, FSDs never deal with the internals of a DCB; it's a data structure used only by IOS and the lower-level device control software.

One of the fields in a DCB is the address of the calldown chain for the device. IOS's successive passing of pointers to the appropriate DCB and IOP to each entry in the device calldown chain defines the path of execution within IOS and its lower-level driver modules.

Calldown Chains

The multiple layers of the filesystem architecture offer a great deal of flexibility to device driver writers. Essentially, you can get control at any point in the path between an application's issuing a file-related API and the lowest-level device driver's poking the controller registers. This flexibility is a far cry from the single INT 21H hooking technique practiced by existing MS-DOS filesystem and device control software.

The calldown chain technique is what Windows 95 uses to implement the multilayer mechanism. During initialization, a device driver module can add itself to the calldown chain for a particular device, specifying the level for the subsequent call. (This is similar to the technique for specifying the initialization level for the device driver module.) IOS inserts the address of the target function into the calldown chain for the device—using the specified level to order the chain correctly. As an I/O request proceeds from IOS down to the hardware, IOS arranges to call each function in the calldown chain for the device.

A driver routine inserted in a calldown chain may elect to pass the request on—either unmodified or not—to the next lower layer, or if

able, the routine may simply complete the request and never pass it on down the chain. A driver can also arrange a callback on completion of a device request by the next lower layer. This amounts to a feature equivalent to the calldown chain, but the call occurs after the device operation rather than before.

Asynchronous Driver Events

Asynchronous events notification allows IOS to interact with device driver modules outside the flow of normal I/O requests up and down the driver hierarchy. In some cases, the driver itself asks IOS to signal an asynchronous event at some later time. In other cases, IOS initiates the request.

IOS signals an asynchronous event by calling the driver's asynchronous event entry point, passing it an *asynchronous event packet,* or *AEP.* An AEP has a standard header that specifies the asynchronous function and the associated device data block (DDB). The AEP also has a field the driver uses as a completion code. Beyond the header, the structure of the data block differs according to the type of event and contains additional event-specific parameters. Here's a summary of the function of each asynchronous event that IOS can signal:

AEP_INITIALIZE	Initialize the driver. Sent when a driver is first loaded.
AEP_BOOT_COMPLETE	System boot is complete. The driver can switch to its runtime configuration.
AEP_CONFIG_DCB	Configure the physical device and associated DCB.
AEP_IOP_TIMEOUT	Time-out counter within an IOP has reached 0.
AEP_CONFIG_LOGICAL	Configure the logical device.
AEP_DEVICE_INQUIRY	Retrieve device identification information.
AEP_RESET_COUNTERS	Reset performance counters.
AEP_REGISTER_DONE	Registration processing is complete.
AEP_HALF_SEC	Half a second has elapsed.
AEP_1_SEC	One second has elapsed.
AEP_2_SECS	Two seconds have elapsed.
AEP_4_SECS	Four seconds have elapsed.
AEP_DBG_DOT_CMD	Pass debug parameters to the driver.

Interfacing to the Hardware

Port drivers are the most common manifestations of the hardware control level in the filesystem software hierarchy. Port drivers that control ISA or EISA configuration adapters interface directly to the hardware. In the absence of another intermediate layer, such as a volume tracking driver layer or a protected mode BIOS layer, the type specific driver (TSD) provides the only other software layer between IOS and the hardware. The port driver is, therefore, what you would typically think of as the "device driver" for the filesystem.

The port driver is hardware specific, and although layers such as the TSD's reduce the driver's workload, the port driver still has the job of translating I/O requests into hardware commands. As with the development of most drivers for devices with similar characteristics, developing a new port driver will typically involve the modification of an existing example rather than the creation of entirely new code. A port driver is a dynamically loaded VxD that provides no VxD services. Figure 7-8 illustrates the declaration of a port driver together with the driver registration packet (*DRP_Port*) used by IOS during the driver's initialization phase. Notice the inclusion of the pointers to the port driver asynchronous event routine (*PORT_Async*) and to the ILB structure (*PORT_ilb*) that IOS needs to complete the initialization process.

```
DECLARE_VIRTUAL_DEVICE PORT, MAJOR_VER, MINOR_VER,
    PORT_Control,, UNDEFINED_INIT_ORDER,,, DRP_Port

DRP_Port DRP <EyeCatcher, DRP_MISC_PD,
    offset32 PORT_Async,
    offset32 PORT_ilb,
    PORTname, PORTRev, PORTFeature, PORT_IF>
```

Figure 7-8.
Port driver and DRP declaration.

We've already looked from the IOS perspective at what happens within the filesystem hierarchy. Turning this around, let's look at a summary of an individual port driver's responsibilities during different execution phases.

Initialization

IOS first sends a SYS_DYNAMIC_DEVICE_INIT message to call the port driver. The port driver uses the *IOS_Register()* service to register

itself. During registration, the port driver has to respond to callbacks to its asynchronous event routine:

AEP_INITIALIZE requires allocation of a DDB, retrieval of configuration information, initialization of the hardware, and definition of the device's interrupt handler.

AEP_DEVICE_INQUIRY messages are sent for each possible drive attached to the adapter. (The design accommodates drive numbers 0 through 127.) The port driver must respond with an indication of the presence or absence of a particular drive.[12]

AEP_CONFIG_DCB allows the driver to add its normal I/O request entry point to the calldown chain.

AEP_BOOT_COMPLETE allows the port driver to confirm or deny that it has detected hardware it can control. IOS will remove the driver from memory if no applicable hardware is present in the system.

Execution

Normal execution for the port driver involves processing and queuing IOPs passed to the driver via its normal I/O request function. For actual device I/O operations, if the device isn't busy, the port driver starts the operation. The port driver must also respond to time-out events (AEP_IOP_TIMEOUT) signaled by IOS.

Interrupt

If the device interrupts as the result of its completing an I/O operation, the port driver finishes processing the associated IOP. If there are other IOPs queued for the device, the driver starts the next I/O operation.

Other Layers in the Filesystem Hierarchy

Of the other available levels within the IOS managed hierarchy, a few are used by components that are standard modules within the Windows 95 filesystem architecture. In general, the modules installed at these intermediate levels are designed to provide services commonly required by port drivers. The installation of these modules relieves a

12. The port driver can also respond with an indication of *no more devices present* to avoid processing 128 separate inquiries.

driver developer from having to re-implement private versions of functions needed by every driver. The type specific driver (TSD) for disks, for example, will perform some error checking and logical to physical parameter translation, relieving the individual port drivers of this chore. Supplying standardized components such as these is also a means for Microsoft to avoid problems with device driver bugs. The more complex a single device driver, the more likely it is to contain bugs and the more likely it is that Microsoft's technical support group will get a phone call. A user will regard the problem as a bug in Windows—rare is the user who would call Exotic Disk Drive, Inc., if Windows crashed because of a bug in the device driver that came with the drive.

The most highly developed use of the IOS layering capabilities is for the support of SCSI devices. Microsoft Windows NT placed a lot of emphasis on the support of SCSI peripherals—partly because the market for these devices was growing rapidly during the development of Windows NT and partly because SCSI peripherals were a good match for the Windows NT performance and automatic configuration goals. The SCSI design also standardizes many device interface issues, making SCSI devices a perfect match for the layered device architecture.

Windows 95 standardizes other existing features of block device drivers by including modules that manage the issues associated with exchangeable media and by providing a generalized interface to data caching. New in Windows 95 are the support for Plug and Play capabilities and the continued support of real mode device drivers within a fully protected mode operating system.[13]

Volume Tracking Drivers

The volume tracking driver, or VTD, is at the top of the calldown chain for a device. Its role is to ensure that the medium in a particular drive (usually a floppy disk or a tape) is the medium that the I/O request actually refers to. Obviously, in the case of a read operation, a medium that doesn't match what the application previously referred to will probably be only confusing to the user; in the worst case, though, the mismatch could cause an application to fail. In the case of an output operation, the effect of writing on the wrong medium could be disastrous.

13. Windows NT ducked this particular challenge by not providing MS-DOS device driver support. Given its compatibility requirements, this was not an option for Windows 95.

The VTD maintains its knowledge of the current medium by matching a volume handle retained in the current DCB for the device with the volume handle contained in any IOP passed down by the filesystem driver. A mismatch means that the medium present in the device is not the medium previously referred to by the FSD. This may result in the user's being asked to insert the correct medium.[14]

Knowledge of the current volume is maintained by the IOS's asking an FSD to read a volume label each time the medium changes (an event that the device driver will notice) or, in the case of hardware that can't report a medium change directly, whenever the medium may have changed. It is up to the FSD to read volume labels because the other components of the filesystem have no knowledge of how to do this. The FSD retains information about a volume label from the time the medium is first mounted.

Type Specific Drivers

In Windows 95, a type specific driver (TSD) currently exists for a disk device in order to provide a mapping from logical to physical device parameters. Using handles and offsets, an FSD will typically translate application requests to requests for logical block numbers within a logical drive—for example, *read block 93 of drive C:* (where a numeric handle would represent C:). The TSD will translate such requests for logical block numbers into physical block numbers. This translation may involve a mapping of logical blocks into physical blocks (where the device's sector size doesn't match the filesystem's block size) or the translation from a logical drive to a specific physical disk partition. The TSD checks every request it processes, ensuring that the lower-level drivers don't have to perform any validation.

During initialization, the TSD is responsible for allocating and building a device control block for each logical device present on a physical device (for each hard disk partition, for instance). The TSD adds each logical DCB to a list that is associated with the DCB previously allocated to describe the physical device. Within the logical DCB is all the information describing the geometry of the drive device—sectors per track and bytes per sector, for instance.

14. Volume tracking requirements may change according to the environment. If a file is left open with data still to write out, a different medium is usually an error. For a multivolume backup operation, though, it's an expected condition.

One valuable contribution to flexibility this architecture affords is the ability it gives the system to adapt to the different geometry on high-capacity exchangeable media. Several manufacturers now offer drives with removable media that can store 100 megabytes of data or more. Most of these drives can read older, compatible but less densely packed media. The Windows 95 filesystem participation in the dynamic reconfiguration of the device characteristics for specific partitions helps to support these devices properly.

SCSI Manager

Windows 95 builds on the SCSI device architecture developed for Windows NT by making use of the same low-level device drivers (the so called *miniport* drivers). By providing a method for interfacing existing Windows NT miniport drivers to the Windows 95 filesystem architecture, Windows 95 gains immediate support for a wide range of SCSI peripherals with almost no new code having to be developed. This method for interfacing drivers between the two systems is another manifestation of the Windows compatibility goal. For a device manufacturer, the fact that a single miniport driver will support two different operating systems is a definite benefit.

The SCSI manager, or SCSI port driver, is the upper layer of this support. The SCSI driver offers a range of functions common to any SCSI device, including error logging, cache management, and logical to physical address translation. Essentially, the SCSI manager and the miniport drivers associated with it split the functions of a normal port driver, with the hardware-specific aspects isolated in the miniport driver. Three main data structures are used for communication between the SCSI manager and the miniport drivers:

SCSI_REQUEST_BLOCK contains information describing an individual SCSI device I/O request.

HW_INITIALIZATION_DATA contains the miniport device driver's entry points called by the SCSI manager for a specific device.

PORT_CONFIGURATION_INFORMATION contains data that describes the properties of an individual SCSI host adapter, including, for example, its DMA capabilities.

The entry points provided by each miniport driver allow the SCSI manager to call for hardware-specific operations during various phases of

device control: initialization, I/O request initiation, and interrupt processing.

For the ultimate efficiency in implementation, use of the existing Windows NT miniport drivers would have been the optimal solution. Unfortunately, the requirements of real mode compatibility made their presence felt once again. The existing miniport drivers for Windows NT have to undergo a few minor modifications for full compatibility with Windows 95. The modifications have largely to do with the real mode to protected mode transitions and with the fact that, in Windows 95, a real mode SCSI device driver can exist in conjunction with the protected mode miniport driver. However, once the driver has been modified to accommodate the need for real mode compatibility in Windows 95, the new version will still run under Windows NT—the new real mode support code will simply never be executed in the Windows NT environment. Note too that as with support for any device under Windows 95, SCSI drivers should participate in the Plug and Play environment and that means other modifications to the miniport driver.

Real Mode Drivers

Continued support for existing MS-DOS real mode device drivers is obviously critical to the success of Windows 95. Despite the advantages of protected mode device drivers, the sheer number of drivers available for MS-DOS means that it will be impossible to replace every real mode driver when Windows 95 first ships. But replacement of the real mode drivers for many widespread devices, such as IDE hard disk controllers and NEC-compatible floppy disks, will happen immediately, so most users will quickly see the performance benefits of the new protected mode filesystem.

The filesystem design in Windows 95 allows a protected mode port driver to take control of a real mode driver and bypass it while the system is running in protected mode—Windows 95 can classify the real mode driver as a "safe" driver, that is. Safe means, essentially, that the protected mode driver can offer functionality identical to the real mode driver's. In such a case, the protected mode driver will simply carry out all the I/O operations and never call the real mode driver. In a number of instances, the protected mode driver's taking over the function of the real mode driver is considered unsafe. The real mode driver may do data encryption, for example, or may interface with a real mode system BIOS to do dynamic bad sector mapping for the hard

disk. The standard Windows 95 port driver for the disk adapter, though able to control the hardware, can't replicate this extra functionality, so it arranges to route I/O requests through the real mode driver—executing the driver in virtual 8086 mode in order to do so.

To recognize a safe driver, Windows 95 maintains a list of such device drivers by means of the registry. If the system running in protected mode detects the presence of a real mode driver, it consults the safe driver list to determine whether the real mode driver functions can be subsumed under the protected mode driver functions. The identification for the real mode driver is its name as entered in CONFIG.SYS or AUTOEXEC.BAT. If the driver name doesn't appear in the safe driver list, Windows 95 will use the real mode driver.

Conclusion

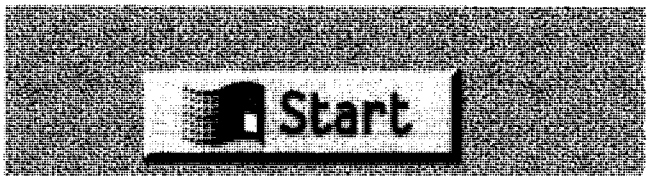
From the discussion in this chapter, you've no doubt realized that the new filesystem design for Windows 95 is a major revision to Windows. Although the compatibility constraints imposed on Windows 95 allow a device manufacturer to continue to support hardware using an older real mode device driver, the advantages to be gained in terms of performance, multitasking, and reduced memory requirements are compelling reasons to provide a full Windows 95 protected mode driver. And, of course, the addition of long filename support is a huge benefit to the user.

The new Plug and Play subsystem augments many of the operations of the filesystem components, and that's what we'll look at in the next chapter. The installable filesystem capabilities also dramatically improve networking support in Windows 95, and that will be the subject of Chapter Nine.

References

Microsoft Corporation. Windows 95 Device Driver Kit. Redmond, Wash.: Microsoft, 1994.

Schulman, Andrew. *Undocumented DOS*. 2d ed. Reading, Mass.: Addison-Wesley, 1993.



CHAPTER EIGHT

PLUG AND PLAY

If you've ever had to suffer through the experience of opening up a PC system unit to plug in a new device adapter card, you'll immediately understand why Plug and Play is important. The combination of Windows 95 and a PC that supports the Plug and Play specification will reduce your system setup and reconfiguration suffering to a minimum. You'll still have to know how to use a screwdriver, but that's about the only extra skill you'll need. Although the collaborators who developed the Plug and Play specification deliberately avoided tying the standard to a particular operating system or hardware type, Windows 95 has the distinction of being the first system to provide full support for the Plug and Play standard.

Typically, the process of adding a new device to a PC has involved figuring out how to set all the switches and jumpers on the new card, plugging the card in, installing software, rebooting the system, and praying. The amount of time you could spend trying to resolve problems during the installation of a new device could be extensive. Every PC has one or more *bus devices.* Usually, several devices are trying to share the *system bus,* and those attempts to share often lead to conflict. The bus design determines the electrical characteristics of many system components as well as some aspects of the method that device driver software must use to control an individual device on the bus. Most PC buses conform to a specification referred to as *industry standard architecture,* or *ISA* for short. The ISA specification is little more than the formal description of the original IBM PC architecture that was written down long after the PC first went on sale.

Most device adapter cards plug directly into the system bus. The software that controls a device communicates with the adapter by writing commands to the system I/O ports. The command information travels along the system bus to the device adapter. Some devices (often called

memory mapped devices) also use a memory region in the 640K to 1-MB upper memory area. Both the device and the device driver software can access the data in that memory area, allowing for the high-speed transfer of large amounts of information between the device and the system's memory. Non-memory-mapped devices transfer data by means of the system bus, raising a hardware interrupt when they need attention from the device driver.

When you first plug a device adapter into the bus, it is normally set up to communicate with the system by means of a default set of I/O addresses, interrupt requests, and possibly a shared memory region or a direct memory access (DMA) channel. If some other device on the bus is already using one or more of these control signals or memory areas, a conflict occurs. The system will usually react to the conflict by refusing to boot properly, requiring you to open the box again and try to resolve the conflict by selecting a different configuration. Or sometimes the system will boot but the device will appear not to work when you try to access it, calling for more reconfiguration effort. Once you have working hardware, you have to configure the associated software to match. Over the history of the PC industry, this type of configuration activity has probably consumed the lion's share of the effort put forth by technical support groups all over the world.

What's the solution? Automatic management of the system's low-level hardware resources—IRQs, I/O ports, DMA channels, and memory—seems to be the key. Plug and Play is Microsoft's attempt to provide such an automatic system management capability. Full Plug and Play support will appear for the first time in Windows 95 and, Microsoft says, will appear over time in their other operating system products. In Windows 95, the system setup process relies heavily on the Plug and Play system management capabilities. And once the system is up and running, the Plug and Play subsystem is responsible for managing all hardware configuration changes.

Why Do We Need Another Standard?

Naturally, there have been other attempts to solve system configuration problems, but none of them has achieved the critical mass of support that's necessary to truly eradicate the configuration conflict problem. The two best-known solutions each involved the introduction of a new system bus design: IBM's MicroChannel bus, used only in IBM's PS/2 series, and the EISA (Extended Industry Standard Architecture) bus.

The designers of the MicroChannel bus came up with a new bus design that allowed any card plugged into the bus to identify itself to the operating system. After plugging the card into the bus and installing the device software, you could configure the adapter card using a standard configuration program. Unfortunately, the MicroChannel design suffered from a number of problems. First, the MicroChannel bus was incompatible with the existing ISA bus. You couldn't take your old network adapter, for example, and simply plug it into a MicroChannel bus. Since the PS/2 series never came to dominate the market, the MicroChannel never won wholehearted support from other device manufacturers. The other problem with the MicroChannel bus was that every adapter needed a unique identifying number, issued by IBM, that was hardwired into the adapter. This requirement reduced configuration flexibility somewhat, and the user still had to work his or her way through the device configuration program in the event of a system conflict.

The EISA bus designers adopted some of the better ideas in the MicroChannel design but based their design on the ISA bus. The big advantage of an EISA bus was that you could use any existing ISA adapter in an EISA machine, although the smarter configuration facilities were available only for new EISA adapters. Several PC companies ship EISA systems, and the EISA bus has gathered a reasonable amount of support from device manufacturers, but EISA is by no means a dominant architecture either.

Other, perhaps less ambitious, attempts to reduce hardware configuration problems include the efforts of suppliers who preconfigure systems with network cards, pointing devices, and the appropriate software already set up. Microsoft's Windows "Ready To Run" campaign was based on the expectation that PC vendors would ship preconfigured machines with Windows 3.1 already installed. Some device manufacturers allow devices to be reconfigured without anyone's having to open up the machine and reset hardware jumpers and switches. Intel's EtherExpress network adapter is a good example of this type of relatively easy to configure device. You plug in the adapter, and if the default adapter configuration doesn't work, a software setup program allows you to change the hardware configuration with commands from the keyboard.

All of these solutions share some of the shortcomings itemized on the next page.

- There is still no single, generally accepted standard for device installation and configuration. In particular, there is no standard for the market's leading hardware architecture: the ISA bus. A single standard would help by encouraging every manufacturer to adopt the same solution to the problem. A standard that catered to the ISA bus as well could greatly reduce the problems of hardware setup for the majority of users.
- Whereas a PC used to have just one bus, recent technology improvements have led to PCs that incorporate multiple buses: SCSI, PCMCIA, and various types of local video buses, for example. None of the existing configuration methodologies allows for this mixture of bus types.
- There's a growing need for a dynamic configuration method. Consider the situation in which you might have a modem on a PCMCIA card plugged into your laptop as COM1 and you connect the laptop to its docking station, which has a more conventional serial COM1 device. Or consider the dynamic reconfiguration requirements of a wireless-based network that supports mobile workstations. None of the existing solutions is flexible enough to handle this kind of situation.

The Plug and Play standard tries to address all of these issues, and Windows 95 intends to be the first major operating system to provide full support for the Plug and Play standard.[1]

History of the Plug and Play Project

The Plug and Play standard has its beginnings in the several different attempts to address the problem of hardware configuration—with IBM's Micro Channel and the Extended Industry Standard Architecture (EISA) effort initiated by Compaq among the most well known. Microsoft's Plug and Play effort began in 1991, and the first public specifications appeared during 1993.[2] At first, Microsoft worked on the

1. For information about the pieces of the Plug and Play specification, see the "References" section at the end of this chapter.

2. Folklore has it that the initial impetus for the project was provided by the PC configuration problems experienced by the mother of the vice president of Microsoft's Personal Systems Group. Another story cites Microsoft's irritation at the advertising campaign run by Apple Computer—the one that portrayed Windows as hard to set up and use.

specification alone, seeking an ordered solution to an apparently intractable problem. Early discussions with Intel and Compaq helped to steer the design effort, although these companies did not formally agree to support the Plug and Play standard until the spring of 1993.

The deciding factor in wider industry support was the development of the Plug and Play ISA specification—a document that defined a modified hardware design for adapter cards that could be used on existing ISA bus PCs. Also included in the Plug and Play ISA specification was a software-only solution that could be applied to the installed base of "legacy adapter cards" (a new term considered more polite than "old adapters"). These accommodations of the installed base are where the Plug and Play effort differentiated itself from earlier initiatives. Both the MicroChannel and the EISA bus designs did little to help the users of the installed base of PCs. The attention it paid to the predominant ISA bus design moved the Plug and Play effort from a somewhat academic realm into the entirely practical world. And the fact that a Plug and Play compliant adapter card could be produced for only a tiny amount more than it cost to produce existing adapters made the Plug and Play specification immediately attractive to a broad range of manufacturers. (Microsoft had started with a cost target of a few dollars and realized early on that this would be too expensive. Current estimates pin the hardware cost of adding Plug and Play at around 25 cents.) Once the Plug and Play ISA specification was out, support for the standard gained momentum during 1993, with Intel supplying early developer kits, Phoenix Technologies joining the core group to help define a new BIOS for Plug and Play systems, 3Com providing extensive technical input, and companies such as Future Domain releasing early ASIC implementations of the Plug and Play hardware interface.

By the end of 1993, variants of the Plug and Play specification had been produced for several different bus types, including the ISA, PCMCIA, PCI, and SCSI types.[3] The Plug and Play effort began to have other influence as well. Inside Microsoft, the design of the Windows NT registry underwent modification to incorporate Plug and Play capabilities before the shipment of Windows NT. Outside Microsoft, design efforts such as the IEEE's serial SCSI specification began to take Plug and Play requirements into account.

3. This effort continued, and specifications for every major bus type (except EISA) and for several specific devices (such as the parallel port) had been produced by mid-1994.

At the time of this writing, the Plug and Play effort has a long way to go before a complete implementation will be in the hands of a large number of users. Microsoft gained early experience with some of the device detection and configuration techniques they deployed in products such as Windows for Workgroups and Windows NT. These systems try to automatically sense the configurations of their host machines. In the case of Windows for Workgroups, it's the video adapter, mouse, keyboard, and network adapter types that the operating system tries to figure out. Windows NT goes much further, sensing SCSI devices and other installed hardware. The benefits during installation are obvious. Windows 95 goes further still, implementing almost automatic installation and dynamic reconfiguration. Regardless of the success of Windows 95 itself, the Plug and Play specification certainly seems to have enough momentum to gain real acceptance in the marketplace.

Goals for Plug and Play

The Plug and Play project identified a number of goals that the specification, and any of its implementations, needed to meet. The overriding goal, though, was simply to make it easier to add new hardware to or change the configuration of an existing system—actually, not just easier, but very, very easy. This ease helps everyone. Users waste less time and get less frustrated when they try to change their hardware. There's less burden on any support groups that users might call. The device manufacturers have a well-specified standard to develop to rather than the prospect of trying to solve all the potential installation and configuration issues themselves. With new hardware developed to the Plug and Play standard, the goal of requiring absolutely no effort beyond plugging in the device and copying the software to the hard disk can be realized. With existing hardware, it's difficult to reach that level of simplicity because the hardware itself doesn't conform to the Plug and Play standard. However, a lot can be done in software alone, and the Plug and Play standard calls for upgrades to existing device driver software. Upgraded device driver software will allow current ISA hardware to be well managed within a Plug and Play environment.

The Plug and Play specification lists five formal goals:

- Easy installation and configuration of new devices
- Seamless dynamic configuration changes

- Compatibility with the installed base and old peripherals
- Operating system and hardware independence
- Reduced complexity and increased flexibility of hardware

Plug and Play is of course the core of one of the major goals for the Windows 95 project: great setup and easy configuration. And the specification's attention to the existing ISA hardware base is a necessary aspect of the compatibility goal set for the Windows 95 product.

Let's look briefly at each of the major Plug and Play goals.

Easy Installation and Configuration of New Devices

With new—that is, full Plug and Play specification—hardware, the installation and configuration process is reduced to plugging in the device and running a simple installation program. Some assembly is required, but the installation program does little more than copy the device support software to the Windows directory. During the boot process, the system can identify the device and locate the appropriate device driver software and load it. The responsibility for identifying the hardware devices and configuring them correctly belongs to the operating system, not the user.

For the reasons we've already reviewed, the Plug and Play standard provides a potential for tremendous savings of time and effort. The drawback is that for the full Plug and Play benefits to be realized, you need a full Plug and Play machine and full Plug and Play device adapters.

Support for a New Hardware Standard

The Plug and Play specification does not define yet-another way of building a PC. What it does specify is what PC hardware must be able to do if it is to support full Plug and Play capabilities. "PC hardware" means the system motherboard, the BIOS, and the plug-in adapter cards. If each of these components complies with the specification, the operating system vendor can implement Plug and Play. To date, draft or final specifications have been completed for the Plug and Play BIOS and for the ISA, SCSI, PCMCIA, and PCI buses. By the time you read this, there will be many other specifications for Plug and Play compliant hardware.

Some current bus designs lend themselves to a very simple implementation of Plug and Play support; the required information and capabilities already exist. All that's needed is the appropriate layer of

software to provide the information in Plug and Play format. For the existing ISA bus, implementation of Plug and Play support is a lot harder. However, the low-level operations that the bus and associated devices must support are somewhat similar in every case:

- Isolating a device. There has to be a way for the operating system to interact with one, and only one, device at a time during the system boot process. If two devices respond to the same operating system inquiry, the process breaks down.
- Reading information from the device. The Plug and Play subsystem needs to collect information from the device. For a Plug and Play device, a defined interface allows the device to provide specific information in a standard format. In the case of a legacy adapter with no provision for Plug and Play support, the software has to collect whatever information it can and then play the software equivalent of a word guessing game during the identification step.
- Identifying the device. Whatever information the device provides must be sufficient for the Plug and Play subsystem to correctly identify the device. Identifying a 3Com network adapter as a Hewlett-Packard scanner will obviously lead to problems.
- Configuring the device. Plug and Play devices expect to be told which resources they can use: which IRQ, which I/O ports, which DMA channel, and which memory region. This provision is a key aspect of the Plug and Play specification design. No longer will you enter a deadlock situation in which two different devices absolutely require use of the same IRQ. Non–Plug and Play devices don't have a reconfiguration capability, so the resources these cards consume are reserved first and made unavailable to other devices.
- Locating and loading a device driver for the device. Once the device driver is loaded, it takes over the control of the device, using the allocated resources.

Devices that conform to the full Plug and Play specification make the operations in this process quite straightforward. The various specification documents for Plug and Play hardware describe the requirements

and implementation methods in great detail. The more difficult job is making the legacy cards appear to behave like Plug and Play devices.

New ISA Board Standard

Since ISA systems are what most of us own, it's interesting to take a brief look at how the Plug and Play specification augments the ISA adapter design so that ISA systems can support full Plug and Play operations. The Plug and Play specification describes all the hardware and software components in elaborate detail. Essentially, a Plug and Play ISA card must include a small amount of additional hardware logic that implements the following sequence of behavior:

1. At power on, the device remains quiescent until it senses a specific pattern of commands written to a predefined I/O port—the so called *initiation key.*
2. The device then enters a state in which it waits for a "wake" command written to an I/O port. In response to a wake command, the controlling software can either wake up a specific card, if it already has a unique identifier for the card, or move all the cards to the "isolation" state.
3. The Plug and Play software communicates with one and only one card in the isolation state. The device responds to commands sent via the I/O ports by sending data bytes back to the Plug and Play software. The data the device sends back includes a unique identifier that allows the software to identify the device—the identifier includes fields such as a manufacturer ID to ensure unique identification.
4. Once the device has been uniquely identified, the software and the device can exchange information. In this exchange, resource requirements are identified and allocated.

For the cost of redesign and a small increment in manufacturing overhead, an existing ISA card can become a Plug and Play device. Preferably, the host system will have a new Plug and Play BIOS and of course a Plug and Play capable operating system such as Windows 95.[4]

4. The Plug and Play BIOS ensures that a system with multiple boot devices will in fact boot. However, if a Plug and Play BIOS is not present, the operating system takes over all the device configuration chores.

Seamless Dynamic Configuration Changes

With this rather grandiose phrase, the Plug and Play standard addresses the increasingly common situation in which a system's hardware configuration changes while the system is running. No, you won't be opening up your desktop machine and plugging new cards in while your C compiler runs, but there are already a lot of systems available that do allow hardware configuration changes while the system continues to run. The currently popular example of this capability is a laptop system that supports the PCMCIA peripheral standard. Other examples include infrared printer connections and wireless-based networks. The hardware specification for PCMCIA cards took quite a while to develop to everyone's satisfaction, but now a wide variety of PCMCIA-standard peripheral devices is available. In addition to the attractions of their small physical dimensions and light weights, these cards allow you to alter a system's configuration by simply removing one card and plugging in another. You might use an Ethernet card connected to the office network, for example, and exchange it for a fax/modem card while you're traveling. During 1993, many more of the manufacturers began to offer systems with PCMCIA slots, including PCs that use nothing but PCMCIA card slots, such as Hewlett-Packard's OmniBook.

Obviously, the convenience of PCMCIA, or other dynamically reconfigurable systems, is lost if users have to go through an extended software reconfiguration process and reboot whenever they change peripheral cards. The Plug and Play standard addresses this sticking point by defining how a system should allow for hardware resources to be both removed and added while the system is operational. Managing the removal process is easily as important as dealing with the addition of new devices. You certainly don't want the user pulling a disk drive out of the system before all the files on the drive have been correctly updated and closed. Windows 95 takes this aspect of Plug and Play to its logical conclusion by having a notification system inform applications of configuration changes. Every significant configuration change causes a message broadcast that applications can either process or ignore. A facsimile application, for instance, can process a message informing it that the user has tried to eject the fax/modem card. The application's response to the message might be putting up a dialog indicating that there are fax messages still to be sent.

Compatibility with the Installed Base and Old Peripherals

Perhaps the most difficult goal for the Plug and Play consortium to realize was the incorporation of support for the billions of dollars' worth of hardware already in use. Previous attempts at improving configuration flexibility had largely ignored this issue. Not even the combined might of Intel, Microsoft, and the other Plug and Play partners could wave a hardware wand and suddenly make the old systems fully Plug and Play. It was up to the software developers in the consortium to create that magic. The partners realized that achieving the compatibility goal would probably make or break the success of the entire Plug and Play effort.

A number of software components of the Plug and Play implementation contribute to its support for current hardware. Each component makes the configuration process a little easier for the end user. Naturally, some situations will require the user's assistance. For example, if an adapter can be hardware configured only—by moving jumpers and switches on the card, that is—or if the device driver software can't report the adapter's configuration, Windows 95 will have to ask the user for help.

Over the last few years, Microsoft has built a veritable library of techniques for isolating and identifying different ISA devices, and the great majority of popular devices can now be supported by the Plug and Play subsystem. Inevitably, there will be exceptions. If you happen to be the proud owner of one of the only three Flashbang 9000 network adapters ever made, you're almost out of luck. Almost, but not quite. The Plug and Play specification recognizes the need for a fallback position: ask the user for device configuration information. In Windows 95 this might happen during system setup, or during some future reconfiguration exercise called for when the user has added a new adapter that the Plug and Play subsystem simply cannot recognize. A series of dialogs will lead the user through the process of specifying the device and the resources it requires. Once the device is identified, Plug and Play will store the information in the registry and re-use it the next time the system is turned on.

The Plug and Play implementation tries to minimize such appeals to the user for information by both supporting extensions to the device driver software—so that some reporting is available—and recording the current hardware configuration on disk. If you think of the number of times you've lost the scrap of paper on which you'd written the IRQ

you assigned to the network card when you plugged it in, you'll surely appreciate Windows 95 when the time to add another adapter to the system comes around. In the case of device driver software, a manufacturer can provide some Plug and Play support by simply updating the driver. No hardware changes are needed. Given the fairly efficient driver distribution mechanisms in place—the Windows 95 product itself, the device driver library disk, and bulletin boards—it's reasonable to expect that a lot of manufacturers will try to add basic Plug and Play support to current hardware. And you don't have to have updated device drivers. Even with no changes to the driver, Windows 95 will support the device and do its level best to detect the device and its configuration during installation. All of this will go a long way in making Plug and Play attractive to the installed base.

Operating System and Hardware Independence

Given the collaborative nature of the Plug and Play specification effort, you'd expect the standard to address any hardware or operating system environment. And in spite of competitive issues, the Plug and Play specification does acknowledge the importance of providing a suitable base for future development. After all, the introduction of PCMCIA and local bus systems gathered momentum only recently. And efforts such as the IEEE serial SCSI specification have not yet left the committee room. Few people would be willing to bet that there will be no other fundamental industry developments in hardware interfaces. Given the intensity of competition, we can expect major improvements in operating system technology over the next few years.

All of this demands that the Plug and Play specification be independent of the underlying hardware and software. The basic data structures, naming conventions, and user interface aspects of Plug and Play are defined only to a level that allows a consistent implementation of the specification across different platforms. Specific implementation details are left to the operating system developer.

Reduced Complexity and Increased Flexibility of Hardware

We've looked at a number of the complexities surrounding hardware configuration. As we noted earlier, making hardware configuration easy was the prime goal for the Plug and Play standard. The specification also lists the goal of making hardware "flexible." Meaning what exactly? Flexibility goes back to the goal of reducing complexity. One

of the most frustrating problems with current hardware is resolving conflicts between devices. As we've already noted, for example in Chapter Two's history of the Intel processor, two adapters can't share an IRQ or a set of I/O ports. Yet it's asking a lot to expect users to understand this and be diligent enough to check for conflicts as they add new adapters to their systems. Diagnosing conflicts is also difficult: sometimes the system appears to work fine—until it crashes with no warning and no useful diagnostics.

The goal of increased flexibility really amounts to directing manufacturers to produce hardware that can use a range of different device settings and allow the settings to be chosen by the operating system—not by hardwired jumper and switch settings. In practice, this means that an adapter whose default configuration calls for it to use, say, IRQ 3 can be told by the operating system to use IRQ 10 instead. The user will have provided no input to initiate this change and, in fact, will be unaware of it. Such a requirement for flexibility extends to the dynamic reconfiguration of a system, where the system can instruct a device using a particular configuration to change its configuration in situ. Taken to its logical extreme, this flexibility means that any fully Plug and Play compliant adapter could be plugged into any Plug and Play system and be guaranteed to work. No longer will a user need to dismember a system to disable an existing COM port before installing a new fax card.

Although a lot of the burden for implementing this flexibility falls on the hardware manufacturers, it is also good news for them. Hardware that easily adapts itself to any host configuration is likely to massively reduce the technical support a manufacturer will need to provide. Plug it in and it works—with no series of frustrated phone calls to a support technician who must try to figure out how the user can make the device work alongside the other adapters he or she has already installed. Similarly, the documentation for the product will be simpler, and the installation program for the device driver will be trivial.

The Components of Plug and Play

As we've seen, the goals for Plug and Play are ambitious: easy installation, easy reconfiguration, and on-the-fly configuration changes. What's more, achieving the goals involves a number of different people: the operating system supplier, the system manufacturer, the BIOS developer, and the device vendor. Of course, there needs to be a well-defined set of interfaces and clear divisions of responsibility if the

goals are to be met. The Plug and Play specification approaches the problem of dividing and coordinating the labor by defining a layered architecture for implementation and carefully separating functions into different components. To fully understand how Windows 95 implements the Plug and Play standard, we need to look at the major elements of the subsystem. Figure 8-1 is a representation of relationships among the various components. The description of the components here is, not surprisingly, for the Windows 95 implementation of Plug and Play. Many elements would be the same for a Plug and Play subsystem supported by another operating system.[5]

A number of components, not all of which are shown in Figure 8-1, collaborate in the Plug and Play subsystem. Here's a summary of the role of each:

- Hardware tree. The database of information describing the current system configuration. The hardware tree is built by the configuration manager and kept in memory. Every node on the hardware tree is termed a *device node* and contains the logical description of either an actual device or a bus device.
- .INF files. A collection of disk files containing information about particular types of devices. SCSI.INF, for example, holds information about every known SCSI device. During the installation of a new Plug and Play device, a new .INF file specific to that device will be used to help complete the software installation. Usually the .INF file will be on the installation diskette that comes with the device.
- Registry. The Windows 95 registry containing as a subtree the hardware tree describing the hardware.
- Events. A set of APIs used to signal changes in the system's current configuration. In Windows 95, the message system is used to signal events. In other implementations, an operating system component could be used to signal events.
- Configuration manager. The component responsible for building the database of information describing the machine's configuration (in the registry) and notifying the

5. Windows 95 also uses the Plug and Play subsystem extensively during system setup and subsequent device installation. Other Plug and Play–supportive operating systems may do things differently.

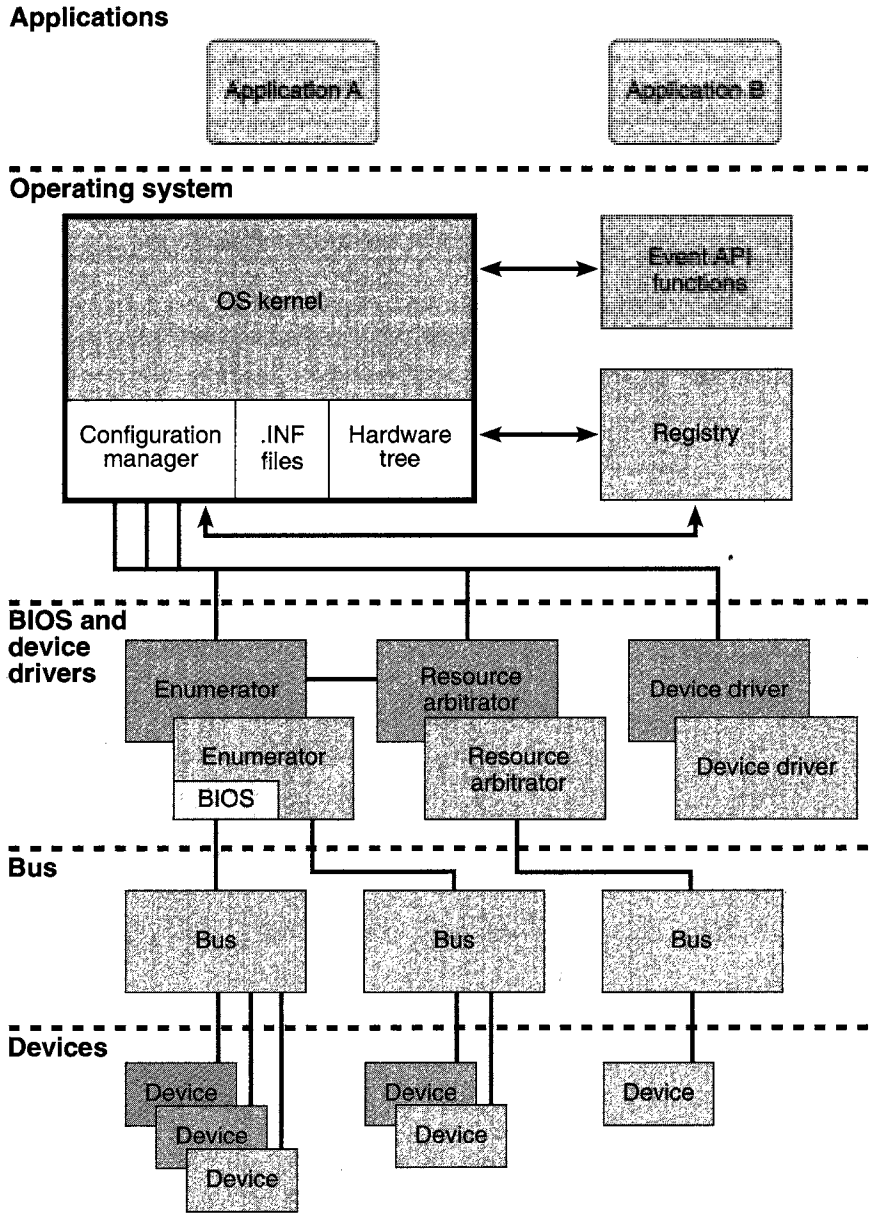


Figure 8-1.
The Plug and Play components.

device drivers of their assigned resources. The configuration manager is the central component of the Plug and Play subsystem when the system is running.

- Enumerator. A new piece of driver software that collaborates with the device driver and the configuration manager. An enumerator is specific to any device (typically to a bus) to which other devices can be attached.[6] Every bus device in the hardware tree always has an enumerator associated with it. A special enumerator, called the *root enumerator*, is part of the configuration manager. The root enumerator assists in setting up non–Plug and Play devices.
- Resource arbitrator. A function responsible for presiding over the allocation of specific resources and for helping to resolve conflicts.
- Plug and Play BIOS. A new system BIOS that supports Plug and Play operations. A device (a video controller, for example) may also have a device-specific BIOS that conforms to the Plug and Play rules. The Plug and Play BIOS is also the enumerator for the motherboard devices and in this guise plays a critical role in managing the docking and undocking operations of portable systems.
- The Plug and Play device drivers. Protected mode drivers responsible for device control as well as participation in the Plug and Play subsystem.
- User interface. A collection of standard dialogs used to solicit information when the Plug and Play system needs to get the user involved in configuration information gathering. The user can also examine the system configuration built by the Plug and Play subsystem.
- Application. In the Plug and Play context, a program modified for improved operation under Windows 95 that can accept and process system configuration change messages.

6. Early designs of the Plug and Play subsystem also used the term *bus driver*. Differentiating the roles of enumerators and bus drivers became sufficiently hard that the functions were finally combined.

Remember that the entire Plug and Play subsystem is mainly concerned with the management of four different resource types on behalf of the various devices:

Memory. The physical memory requirements of the device—for example, how many pages of memory the device needs and any alignment constraints.

I/O. The I/O ports the device will respond to. The device configuration information includes a specification of each of the alternative sets of ports that the device can use (if any).

DMA. Any DMA channels the device requires and any alternative channels it can use.

IRQ. The device's IRQ requirements, alternative IRQs, and whether the device can share an IRQ.

How the Subsystem Fits Together

As you can probably guess, the entire Plug and Play subsystem is a lot of C and assembly language code. Fortunately, very little of the code is memory resident and the system will load most components dynamically. Before we look at the detailed operations of a few components, let's take a step-by-step look at how the whole subsystem hangs together. Central to the entire Plug and Play subsystem is the *hardware tree* data structure that describes the current system hardware configuration. We'll look at the hardware tree's components in more detail in the next section.[7] Figure 8-2 on the next page shows the hardware tree structure that corresponds to a typical Plug and Play system.

Although in this example we're fortunate enough to own a real Plug and Play system, we have held onto our legacy network adapter. Although the network adapter is physically plugged into the ISA bus, as a non–Plug and Play device the adapter is logically attached to the root of the hardware tree during system configuration. More on this in a moment. We haven't made any system configuration changes since the last time we used the system. Let's turn our system on and see what happens.

7. This simple logical representation of the hardware appeared very early in the software design process and has survived every challenge and attempt at improvement.

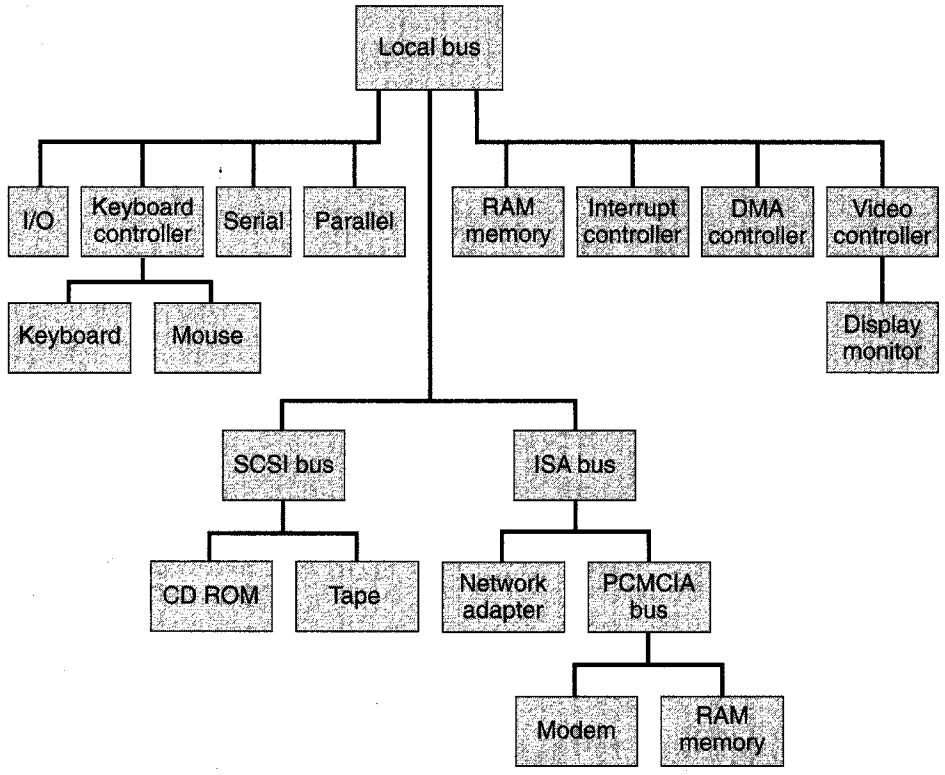


Figure 8-2.
Hardware tree for a typical Plug and Play system.

1. The system BIOS reads nonvolatile memory to determine the machine configuration. The BIOS configures any adapter for which it finds configuration information, notably the motherboard devices. The BIOS disables any adapter for which there is no configuration information.
2. The boot process begins. The system is still in real mode. The configuration manager's root enumerator uses the hardware subtree in the Windows registry as its reference for what the system configuration ought to be.
3. The root enumerator scans the registry subtree looking for all the non–Plug and Play devices. When it finds one, it constructs a device node and adds it to the root of the memory resident hardware tree. This is where you can see the device

node for the legacy network adapter in Figure 8-2's example. The root enumerator also configures the device if the BIOS has not already done so.

4. The real mode boot process continues. The system loader processes SYSTEM.INI, loading the static VxDs that it specifies.
5. Now the next enumerators get loaded. The BIOS has registered the fact that, for example, the system includes an ISA bus. The registry shows which enumerator to load for the particular bus device.
6. The enumerator examines the devices attached to the bus and loads either a static VxD (if one is required) or another enumerator to examine a descendant bus. In the example configuration shown in Figure 8-2, the ISA enumerator would load the PCMCIA enumerator.
7. All the real mode drivers and static VxDs are now in memory. The operating system kernel completes its initialization and switches to protected mode.
8. Now the configuration manager runs. Some of the system's devices are fully initialized, and their drivers are loaded. Other devices simply have their presence on the system recorded with no device driver yet loaded.
9. The configuration manager loads the appropriate remaining enumerators. These enumerators in turn examine the attached devices, build device nodes, and add them to the hardware tree. When this process is complete, the configuration manager will load the device drivers that correspond to the newly created device modes. (During the process, any configuration conflicts will arise and present themselves for solution.)
10. If an unknown non–Plug and Play device is left over, Windows starts the device install process, which asks the user for help in resolving the configuration. Otherwise, the system is now up and running.

Time passes....

After a System Configuration Change

Suppose you automatically load a fax application whenever you start this system. The application uses the fax/modem card on the PCMCIA bus. At some point, you decide you want to transfer the card to another machine, so you press the card eject button.

1. The PCMCIA enumerator receives notice of the button press. It informs the configuration manager. The configuration manager broadcasts the hardware change notification message.
2. Each enumerator sees the change notification message and queries its associated device drivers as to whether they care about your ejecting the card.
3. Eventually, the configuration manager broadcasts a message indicating that the fax card is about to be ejected.
4. The fax application sees the message from the configuration manager and puts up a dialog asking whether you really want to eject the card. You respond Yes. The fax application checks to see whether there are any fax transmissions in progress or pending. If there are no transmissions in progress or pending, the fax application tells the system that the eject operation is OK and returns to a dormant state.

As you can see from this sampling, a lot of interaction goes on among the different Plug and Play components. Much more detail about these interactions would probably overwhelm you. We'll look at a few more implementation details in this chapter, but if you really want every last detail, you need to make the Plug and Play specification itself your favorite bedtime reading.

Hardware Tree

Windows 95 builds the hardware tree during the system boot process, and any subsequent configuration change modifies the tree. The tree is a logical representation of the system hardware configuration. The tree exists as a data structure held in memory while Windows 95 is running. The registry contains a record of every different hardware configuration in the system's lifetime. The memory resident tree is more dynamic, changing as the user adds and removes devices. If you don't change the configuration of your machine from one day to the next, the registry and the memory tree will contain the same (unchanging) information.

Device Nodes

Each node of the hardware tree is called a *device node.* The specification also refers to a node as a *Plug and Play object,* although Plug and Play is not strictly an object-oriented subsystem. The leaf nodes of the tree represent individual devices present in the system—keyboard, monitor, tape, modem, for example. Parent nodes represent *bus devices*—devices that each play a role in the control of at least one other device.

The bus device is fundamental to the design of the Plug and Play subsystem. Plug and Play defines a bus device to be "any device that provides resources." A Plug and Play bus device is also the most common type of parent node for any device node in the hardware tree. In most cases, you can think of the logical Plug and Play bus as the hardware bus in the system. For example, a bus in an ISA system provides interrupt resources (the different IRQs) and I/O port resources. It is also the parent device in the sense that you plug devices into it. In the particular configuration shown in Figure 8-2 on page 326, every node in the tree diagram is a device node, and the *SCSI bus, ISA bus,* and *PCMCIA bus* nodes are bus devices. Take a look at Figure 8-2 again, and note that since the *Keyboard controller* node is also a parent node in the hardware tree, it too is considered a Plug and Play bus device.[8] Every Plug and Play bus device has an *enumerator* associated with it.

Every Plug and Play device node—whether for a device or for a bus device—always contains the following information:

- A unique device identifier—actually a string, not just a number
- A list of resources required by the device node
- A list of resources actually allocated to the device node
- If the device node represents a bus device, a pointer to the descendant device nodes in the tree

Access to the device node data structure is always via a set of system APIs. Device drivers, and other modules, never manipulate the device node data structure directly. Also, it's only the device drivers, enumerators, and other Plug and Play–related modules that use the defined APIs. Application programs never use the APIs.

8.. This is where the mind's eye representation of a Plug and Play bus as a hardware bus breaks down. Thinking of a Plug and Play bus device as *any* piece of hardware that you can plug something into is perhaps a better visualization.

Figure 8-3 is a more detailed representation of a *Network adapter* and a *SCSI bus* device node data structure. The configuration example shown in Figure 8-3 is similar to the example shown in Figure 8-2 except that the network adapter is a Plug and Play adapter.

- All required resources have been allocated to the *Network adapter* node.
- The resources required by the *SCSI bus* child nodes (*Tape* and *CD ROM*) have been allocated.

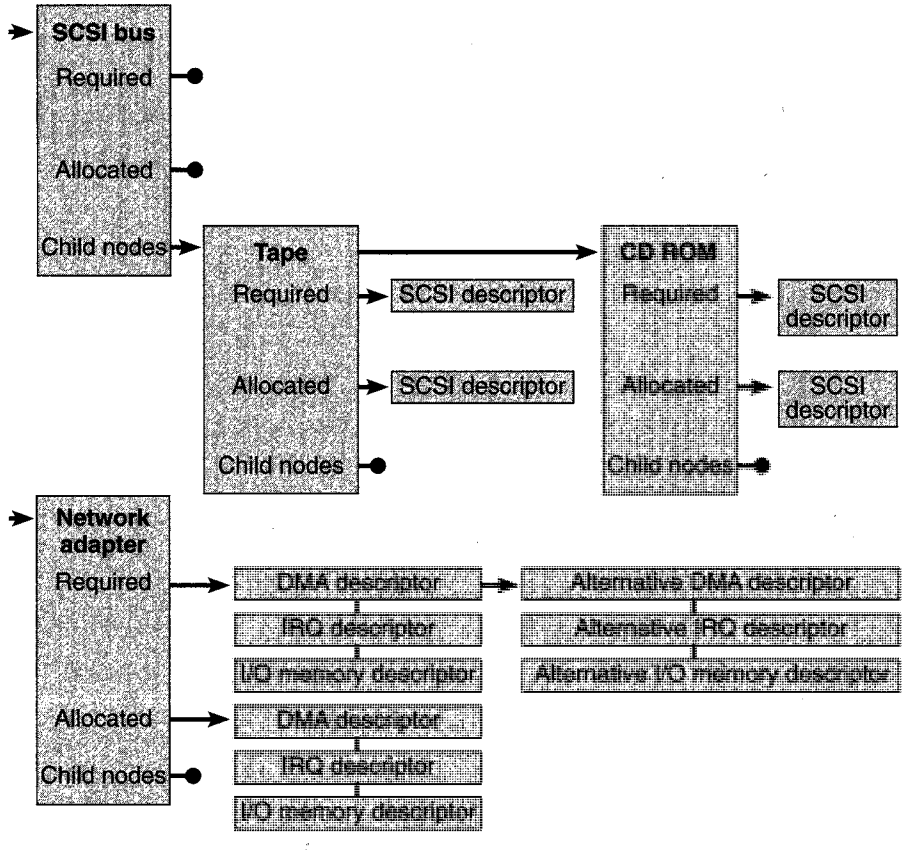


Figure 8-3.
Development of the Network adapter *and* SCSI bus *nodes of the logical hardware tree representation.*

Notice that the *Network adapter* device node depicted in Figure 8-3 has more than one entry in its list for each of the required resources. This provision allows the Plug and Play configuration manager to try to allocate alternative resources when an attempt to allocate an entry in the first set runs into a conflict. For example, if the default IRQ is already in use by another device, the configuration manager will try to use an alternative IRQ. In our example, the registry would have had to contain the information that describes the configuration possibilities for the network adapter.

Device Identifiers

A naming scheme that allows every device on a Plug and Play system to be uniquely identified is a critical requirement for Windows 95. Sensibly, the Plug and Play specification incorporates whatever assistance it can get from currently specified information such as the PCMCIA manufacturer number or the PCI identifier. However, ISA devices have never had a standardized identifying nomenclature, so a new scheme was needed. Rather than trying to evolve an identifier system within the constraints of a 32-bit or 64-bit number, the Plug and Play design uses character strings—sometimes very long character strings. Yes, you can read them, but don't expect to make much sense out of them if you do.

The generation of the device identifier strings is one of the functions of the device enumerator software. The function has to be part of the enumerator since it is this driver alone that is supposed to understand the intimate details of the bus and its attached hardware. Unlike in a static EISA device identifier scheme, the ISA enumerator driver generates the device identifiers dynamically. The algorithm varies from type to type and may involve techniques such as copying company name strings from device ROMs to help. On similarly configured Plug and Play systems with attached ISA devices, the enumerator-generated device name will be the same from one system to the next.

The device identifier for an ISA bus begins with the string *ISAENM*. This beginning at least identifies the enumerator that generated the identifier (and that therefore has control of the device). In our example PC, the modem attached to the PCMCIA bus might end up with a device identifier like *ISAENM\PCMCIAENM\0020071001*—with the trailing digit string's having been generated by the enumerator's reading the manufacturer's ID and part number from the device itself. The enumerator might use just about any naming

scheme that ensures uniqueness. If a system had two identical network cards plugged in, for example, the name string might end with ...*\0300* and ...*\0320* denoting the particular I/O addresses that the cards respond to.[9]

Within the system itself, the device identifiers are very important. Each device node in the memory resident hardware tree contains the device identifier, and the same identifier acts as the registry key the operating system uses to access device-specific information.

Hardware Information Databases

Windows 95 uses four sources of information to determine or record the details of every device on the system:

- The *configuration* files (.INF) held on disk and containing a permanent record of every device ever known. These files arrive already installed on your system.
- The .INF file supplied with each new device (presumably on the installation diskette).
- The user, who has to intercede to solve otherwise unresolvable conflicts or to provide information absent from the databases.
- The Windows 95 registry hardware archive subtree that contains information about the current system configuration. The Windows 95 setup program builds the initial hardware archive in the registry. The registry includes Plug and Play information under three keys:

HKEY_LOCAL_MACHINE	The global settings for the system
HKEY_CURRENT_USER	The current user's personal preferences
HKEY_CURRENT_CONFIG	The current machine configuration—alterable by, for example, whether the system is docked or not

The Plug and Play subsystem draws its information primarily from the hardware archive and the current machine configuration. The user becomes involved only if Windows 95 can't figure out some aspect of

9. Note that the I/O port address is only for identifying purposes. Nothing actually parses the string trying to find the I/O address.

the hardware configuration. Such intervention should come into play only for the older ISA devices that don't conform to the Plug and Play specification.

From all this information, the memory resident hardware tree is built and maintained. Windows 95 updates the hardware tree as the system configuration changes. If you change the configuration before turning the machine on again (switch PCMCIA cards, for example, or replace a defective adapter), the detection process has to refresh the hardware tree with the new configuration.

Note that there is a preferred method of hardware installation for manually configured devices—where you must manually change a jumper setting, for instance. You install the software first, and then you turn the machine off to install the hardware. When you switch the system back on, its configuration will be correctly determined.

Plug and Play Events

Early on in the design of the Plug and Play subsystem, there was a distinct software component called the event manager. Later revisions of the design simplified this notion so that Plug and Play events exist as a set of APIs that use the standard Windows messaging system to allow the broadcasting of messages that describe Plug and Play events. Messages describe events such as requests to remove a device from the system and the addition of new logical volumes to the network. The message from a device driver or enumerator is sent to the configuration manager, which may propagate it on through the system—perhaps in a different form. A device level event in particular could be translated and sent to applications as a window message. Any device driver or VxD can call the event API, specifying the event and providing the associated event data. Applications and drivers with an interest in the particular event will receive and process the message in the normal way.

Configuration Manager

The configuration manager is the principal software component of the Plug and Play subsystem. It's responsible for controlling the hardware tree database and linking the other components of the Plug and Play subsystem together. During the system boot process, the configuration manager is the ultimate authority for ensuring that the hardware tree is fully populated and that its information is correct. The configuration manager is also involved, somewhere along the line, whenever a Plug

and Play event occurs. If a system configuration change occurs, for example, the configuration manager will control the process through which the various bus and device drivers interact, Plug and Play event messages are sent and processed, and modifications to the hardware tree take place.

Here's an example of what happens if a user runs a word processing application, loads a document from a PCMCIA hard disk card, and then presses the card eject button before closing the document file:

1. The PCMCIA disk driver recognizes the card eject button press and notifies the configuration manager.
2. The configuration manager broadcasts the hardware change notification message, which asks whether the card removal operation is allowable.
3. Each device driver responds, indicating that it's OK.
4. The configuration manager broadcasts a message describing the physical device—the hard disk.
5. The I/O subsystem recognizes that the hard disk card contains an active logical drive and broadcasts an application-level message describing the logical device.
6. The word processing application receives the message, processes it, and recognizes that there is a document file open on the affected drive. It displays a dialog for the user that might present two options: save the document and allow the card to be removed, or cancel the card removal and continue.
7. The user's response filters back to the configuration manager in the form of responses to the various messages. In the case of the user's choosing to save the document and thus allow the card to be removed, the configuration manager will ultimately inform the disk driver that the eject operation can proceed. If the user chooses to cancel the card removal, the disk driver will ignore the button press.

Enumerators

An enumerator is a new type of device driver associated specifically with any device that controls another device. Usually, such a device is really a bus, although a device such as the keyboard controller may also

have an associated enumerator. "Enumerator" is an elaborate term for referring to its most common function: walking through each attached device node in its branch of the hardware tree, repeating a particular action. For example, during system startup the enumerator accesses each device on the attached bus, initializing the device and ensuring that the information in the particular device node is complete. The configuration manager calls each enumerator to carry out operations on its attached devices. Using the enumerator this way ensures that the details of the physical bus and the attached devices are hidden from the configuration manager. The enumerator and the associated device drivers deal with the hardware specifics of the device, and the configuration manager deals with device nodes.

The code for a particular enumerator could be implemented by a manufacturer as part of a device adapter BIOS—this is likely, for example, if the system has a proprietary local bus design—or as a protected mode driver that is part of the Windows kernel. For standard hardware, such as the ISA bus, the enumerator is a standard component of Windows 95.

Resource Arbitrators

The other software component that understands the intimate details of a particular hardware device is the resource arbitrator. This kind of function understands the specific hardware resource requirements of a device—for example, the fact that a standard ISA COM device must use either IRQ 3 or IRQ 4. The configuration manager calls an arbitrator function for a device, providing it with the list of required resources from the device node. It is up to the arbitrator to allocate the resources that will satisfy the device's requirements. The configuration manager may also call the arbitrator to inform it that it must relinquish a resource that it is using. Usually, the arbitrator function exists as code within the Windows device driver.

During an attempt to satisfy a hardware resource allocation request, the arbitrator may well come to a dead end. It will need a particular hardware resource, but that resource will already belong to some other device. The arbitrator won't try to resolve the conflict. It will report the error back to the configuration manager and try to provide information that will help the configuration manager resolve the conflict. It's left up to the configuration manager to oversee the process of reallocating resources in an attempt to resolve the conflict.

During this conflict resolution process, arbitrators may be asked to surrender resources they already control. The reallocation process might occur during system startup—the configuration manager reaches a dead end and has to back up—or during a configuration change when a new device requests resources that are already allocated somewhere else.

Plug and Play BIOS

The Plug and Play BIOS is an enhancement of the BIOS that comes in the ROM of every PC. There is a companion document to the Plug and Play specification that describes the details of a Plug and Play BIOS. Every complete BIOS implementation must include both the BIOS functions in use in current machines and the functions that support Plug and Play operation. The design of the Plug and Play BIOS allows both real mode software and 16-bit protected mode software to call BIOS functions. There is no provision for direct calls to the BIOS from a 32-bit protected mode program.

The Plug and Play BIOS extends normal BIOS functionality by

- Maintaining a description of the devices attached to the system board using a data structure very similar to the device node structure used throughout the Plug and Play subsystem
- Supporting a small number of functions that allow an operating system to retrieve and update information about the attached devices
- Providing an event notification mechanism that interfaces with the system configuration manager—this mechanism allowing the operating system to retrieve event information associated with devices that are under BIOS control
- Supporting docking operations on portable systems

The issue of where the BIOS stores the device information is left open to the system and BIOS suppliers. Most systems are likely to use the CMOS memory that the system battery keeps alive. Current PCs already use this memory for storing configuration information, so it's the obvious repository for the Plug and Play information as well. The Plug and Play BIOS specification describes the expected format of the device information that the BIOS must return to the caller. When you make a call to the BIOS function to get device information, the caller

provides a buffer for the BIOS to store the information in. Similarly, when updating the device information for a BIOS controlled device, the operating system calls the BIOS with a modified device node. The Plug and Play specification doesn't allow for direct access to the device information, so exactly where and how the BIOS stores the data is left up to the system manufacturer.

The Plug and Play specification also allows for the BIOS event mechanism to be implemented in two different ways. The BIOS can either simply set a flag in a specific memory location whenever an event occurs or allow the operating system to install an interrupt handler that the BIOS will call to notify the operating system of the occurrence of an event. In the first case, the operating system simply checks the memory location regularly to see whether the event flag is set. Either way, the system must then call the BIOS to retrieve information about the specific event.

Plug and Play Device Drivers

One of the issues facing the Windows 95 team was how to build momentum behind the Plug and Play standard. Although Plug and Play has a broader scope, the fact that Windows 95 would be the first major operating system to support it needed thinking about. Apart from simply convincing all the hardware manufacturers that Plug and Play was indeed a really good idea, the team thought that making it easy to conform to the Plug and Play standard would help a lot. One simple way to make life easy for the manufacturers was to limit the software changes necessary to support Plug and Play. Since Windows 95 can use existing Windows device drivers, you don't absolutely need to develop a new driver to support a Plug and Play system. But this is rather passive support for Plug and Play. To actively support Plug and Play, an existing Windows driver needs to incorporate several modifications and extensions. Here's what such a driver needs to do:

- Be dynamically loadable and unloadable. Thus, a Plug and Play driver becomes a dynamically loadable VxD.
- Use the Windows 95 registry for nonvolatile parameter storage. Windows 95 frowns upon system components that store parameter information in private files or other storage areas. Everything should be in the registry. Information stored under the registry key HKEY_CURRENT_CONFIG also

defines the current machine state—docked or undocked, for example.

- Register with the configuration manager at load time and accept the hardware resources allocated by the configuration manager, and then configure the device according to the configuration manager's allocations rather than according to any existing default.
- Support the release of resources on request.
- Support the new Plug and Play APIs, including the events notified by the event APIs.

The major manifestation of a philosophical change in a Windows 95 device driver is its acceptance of the configuration manager as the controlling entity for resource allocation. Rather than simply initializing a device to a known configuration, the driver must obey the configuration instructions passed to it by the configuration manager. The driver must also respond to event notification if it is to be a good citizen within the overall event system.

Windows 95 device drivers must support several new APIs if they are to operate within the Plug and Play environment. For example, the configuration manager uses specific APIs to either demand or request that the driver release an already allocated resource. Another API tells the driver to configure the hardware according to the resource allocation specified in the device node parameter. The configuration manager may make this call several times while it attempts to adjust the system configuration to avoid allocation conflicts.

Applications in a Plug and Play System

Any application can involve itself in Plug and Play issues by responding to the events that Windows 95 defines. A lot of applications won't care that the system is Plug and Play. After all, a Plug and Play system with no removable devices probably won't change its configuration from power on to power off. However, for many of the latest generation of portable PCs, there are a number of instances in which applications ought to be aware of dynamic configuration changes. Here are a few examples:

- Applications running on portable systems that use PCMCIA cards for disk storage need to take account of the possibility that the user will try to eject a card when there are files open on that disk.
- User alteration of connectivity options—for example, exchanging a network card for a modem card—is likely to be of interest to both the network subsystem and any communications application. The application ought to try to adapt itself to the new speed of the connection, for example.
- Applications ought to adapt smoothly to changes in display resolution initiated by the user.
- The "disappearance" of network volumes when the user walks out of range of his or her wireless network should not result in inelegant or misleading error messages.

In general, applications need to be event aware, and certainly more hardware aware than they have been. Both the new event system and the use of the Windows 95 registry are key to the implementation of standout Windows 95 applications.

Conclusion

In this chapter, we've looked at the Plug and Play specification from the viewpoint of the general goals and architecture of the Plug and Play subsystem. The details we've gone into are specific to the Windows 95 implementation of the Plug and Play specification, but implementations for other operating system environments will share many similarities with the Windows 95 version. If Plug and Play hardware becomes ubiquitous, it's almost certain that other operating systems will support the Plug and Play specification.

Plug and Play represents a major step forward in the ease of use of personal computers. An Apple Macintosh user might assert that they've always had it that good, but then they've also had a much narrower range of third party hardware to choose from. If you remember the theme of Apple's recent anti-Windows television advertising campaign, you'll appreciate how long overdue an enhancement to the PC environment Plug and Play is. Although it will take time for the industry to

catch up and start providing full Plug and Play compliant systems and components, the benefits to users and overwrought support personnel make the effort's wisdom seem compelling.

So far, we've looked at a Windows 95 system from the perspective of a single user. Now that we have Plug and Play, we can fearlessly connect our system to just about anything. The corporate network is probably what occurs first to most of us, so in the next chapter we'll look at the networking capabilities of Windows 95.

References

To receive a copy of the Plug and Play Device Driver Kit (DDK), send electronic mail to *plugplay@intel.com* or fax a request to Intel at (503) 696-1307.

To receive information about future developments at Microsoft on Plug and Play topics, send electronic mail with complete contact information (your name, mailing address, phone number, fax number, and e-mail address) to *playlist@microsoft.com.*

Copies of the various Plug and Play specifications are available for downloading from the Plug and Play forum on CompuServe. Type *go plugplay* at any command prompt. The following specifications are currently available, but others may be added:

The Plug and Play ISA specification

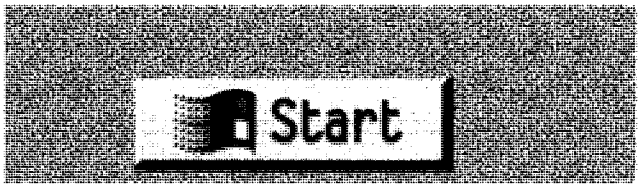
The Plug and Play BIOS specification

The Plug and Play SCSI specification

The Plug and Play PCMCIA specification

The Plug and Play PCI specification

The Plug and Play Advanced Power Management specification



CHAPTER NINE

NETWORKING

Early presentations of the Windows 95 networking strategy characterized Microsoft's goal as "providing the best desktop operating system for networked personal computers." To this end, Windows 95 incorporates full *peer-to-peer* networking capabilities, allowing you to configure self-contained Windows 95 networks with each machine acting as a network server. In addition, Windows 95 aims to provide connectivity to every leading network architecture through a single user interface and a common set of APIs for network applications. Networking under Windows 95 relies on features we've already looked at—most notably on the installable filesystem mechanism discussed in Chapter Seven. In Chapter Ten, we'll look more closely at how Windows 95 handles remote communications; in this chapter, we'll concentrate on Windows 95 support for local area networking.

Although whether or not you'll get networking for free probably won't be clear until the day the product is officially announced, Windows 95 certainly emphasizes networking by incorporating peer-to-peer support, local area network connectivity, and remote connectivity. Windows 95 needed to do a great job of supporting client connections to other networks, and the market positioning for Windows 95 tends to emphasize this connectivity over the peer-to-peer facilities. In fact, most of the newly designed features for Windows networking are more important to client connectivity than to peer-to-peer operation. Microsoft's emphasis on client support is reflected in its development of Novell NetWare support for Windows 95 and its more recent characterization of Windows 95 as "the well-connected client."

Of course, Novell remains the industry's dominant supplier of network products and, at least at the time of this writing, a staunch

advocate of the client-server architecture.[1] The Windows 95 team had to be pragmatic about this situation: their goal that Windows 95 be the perfect client operating system meant addressing the NetWare issue as well as client operation on a Microsoft network. As in the recent release of Windows for Workgroups 3.11, Windows 95 incorporates support for a full Novell client. Buy Windows 95, and you can plug straight in to a NetWare network without buying any other software.[2]

Both Windows for Workgroups version 3.11 and Windows 95 go a lot further than just offering Novell NetWare support alongside support for a Microsoft network. In both products, the system provides for the use of multiple simultaneous network interfaces by using the installable filesystem capability to support remote filesystems. Many users question when on earth they'd ever need to take advantage of this feature. But desktop configurations with, for example, a local link to a NetWare server, a wide area link using a TCP/IP protocol stack,[3] and a dial-up terminal connection to some other network are actually commonplace nowadays. Windows 95 allows these three kinds of network connections to be cleanly integrated—a far cry from the earlier trials and tribulations of networking under Windows 3.0.

Windows Networking History

Before we dive into the technology, let's review some of the history of Windows networking. Microsoft has been an active participant in the network market since 1984, when MS-DOS version 3.1 and MS Net were released. For some years, MS Net was outsold by Novell NetWare, and until the release of Microsoft LAN Manager in 1988, Microsoft really didn't have an industrial strength network operating system. During the same period, network support in Windows was weak—a situation that has changed dramatically as Windows has built its market share over the last three years, since the release of Windows 3.0.

---

1. Novell's acquisition of UNIX System Laboratories and its UNIX technology at least raises the question of whether Novell will ultimately provide a mainstream peer-to-peer network product.

2. Since packaging issues hadn't been decided, in this chapter I've treated "Windows 95" as the networkable version of the product. Maybe the product will be in a single package—maybe not.

3. Basic TCP/IP connectivity was another feature under development for Windows 95 that may or may not be "in the box" for free come product release time.

Peer-to-peer networking has leaped to prominence only in relatively recent times. The release of Microsoft's Windows for Workgroups has sparked a heightened interest in what had been, until late 1992, something of an underground movement in the personal computer industry. When Microsoft announced Windows for Workgroups just before the 1992 COMDEX/Fall trade show, peer-to-peer networking joined the technology mainstream. Despite the apparent youth of the technology, peer-to-peer networks had actually been in wide use since the introduction of the Apple Macintosh in 1985. Apple included the AppleTalk networking capability with each and every Macintosh they shipped. Most early users of the Macintosh were unaware of the fact that they were using peer-to-peer networking whenever they printed a document on the Apple LaserWriter. Apple based the design of the AppleTalk networking protocol on the peer-to-peer principle, and AppleTalk continues to be widely used on Macintosh networks today.[4]

In the PC market, products such as IBM's PC Network and Novell NetWare debuted and began building an installed base. Principally because of the overwhelming success of Novell NetWare, *client-server* networking became known as *the* way to set about connecting multiple IBM-compatible PCs. Microsoft's early network products, MS Net and Microsoft LAN Manager, reinforced the notion that it was a client-server world. In fact, until the release of Windows for Workgroups, Microsoft really didn't acknowledge the existence of the alternative model for networking.

There were companies that had built a business espousing the peer-to-peer model. Products such as 10Net, TOPS, and LANtastic built a solid market base and had many loyal and enthusiastic customers. But it was tough going. On the one hand, they had Apple giving away free networking with every Macintosh, and on the other, they had industry heavyweights such as Novell, IBM, and Microsoft advocating a client-server approach. The companies in the peer-to-peer business found that their products were perceived as suitable only for small networks

4. In keeping with their habit of promoting benefits rather than technology, Apple never pronounced themselves a leader in peer-to-peer networking; nor did they try to promote their technology as the best way to network personal computers. Many users from the IBM-compatible side of the PC universe as a consequence express surprise when they're exposed to the networking capabilities of the Macintosh.

or for small businesses who employed no PC professionals. Although this positioning belied the capabilities of a peer-to-peer network, this type of environment was where the leading peer-to-peer product companies found their easiest sales and their most enthusiastic customers. Competitive pressures have taken their toll on the peer-to-peer network companies, and today only Artisoft's LANtastic has significant market share. Other early products, such as 10Net, have changed ownership a number of times, and although the other peer-to-peer products still exist, they have fairly small installed bases and the future of the various vendors is uncertain. This sad history doesn't sound like much of an advertisement for peer-to-peer networking, but the lack of success so far comes more from the market issues than from any deficiencies in the capabilities of the underlying technology.

Until late 1991, Novell, IBM, and Microsoft continued to espouse the benefits of client-server networking and either ignore or dismiss peer-to-peer solutions. This market situation was an artificial one, created more by marketing dollars than by technology, but it did make good business sense:

- Server software, for use on a more limited number of machines, allowed the supplier to charge a higher price.
- Server-based application software could similarly command a premium price.
- The buyer was often a DP professional, familiar with the client-server model that had been established by the mainframe and minicomputer network manufacturers.
- Network administration tools were often quite poor, even on a server. A peer-to-peer network could compound the problem by putting poor tools in the hands of an unsophisticated user.
- The technology associated with ensuring the security of a peer-to-peer network was still more a research topic than an off-the-shelf product. In contrast, client-server networks provided more reliable security.

Perhaps ironically, the most popular UNIX-based network solutions had also adopted a peer-to-peer model, but IBM-compatible PCs and mainframes remained the stronghold of client-server networking. The situation began to change when Novell introduced its peer-to-peer

product, NetWare Lite, in late 1991. Positioned as a direct competitor to the increasingly popular LANtastic network from Artisoft, NetWare Lite experienced less than spectacular success. NetWare Lite was not a very good product. Novell had tried to ensure that it would not impact upon the continued success of NetWare proper and as a result had introduced a product that was not competitive in its own sphere. The NetWare Lite introduction did put the peer-to-peer concept on many people's radar screens for the first time, however.

In 1992, Microsoft's position on peer-to-peer networking also began to change, as the company began the marketing campaign for its next major operating system product: Windows NT. After years of promotion and successive product releases, Microsoft Windows had become a runaway hit, OS/2 was still selling poorly, and Microsoft had reshaped its plans to promote a Windows operating system product family. At the outset, Microsoft put little emphasis on the networking capabilities of Windows NT. (Remember, Microsoft LAN Manager on OS/2 was the then current solution.) But as more information about the product became available, people began to realize that Windows NT incorporated peer-to-peer networking facilities within the basic operating system. Together with the Windows NT networking news, information about a new version of Windows, called Windows for Workgroups, began to appear. Released for the first time in October 1992, Windows for Workgroups turned out to be a full peer-to-peer network product. During most of 1993, Windows for Workgroups was regarded as a somewhat unsuccessful product, with its critics complaining about slow sales and lackluster features.[5] The "slow sales" charge was unfair; Windows for Workgroups racked up more than a million units in shipments during its first year. And in the fall of 1993, Microsoft released Windows for Workgroups version 3.11—a product that included the debut of a number of features important to Windows 95, such as the protected mode FAT filesystem. Clearly, Microsoft didn't think that peer-to-peer networking wasn't worth further investment. In the summer of 1993, Microsoft had delivered the first production release of Windows NT, with built-in peer-to-peer capabilities, and of course the Windows NT Advanced Server—a product that more closely resembled the client-server architecture of earlier Microsoft LAN Manager releases.

5. Even inside Microsoft, the belief that sales were slow prompted company humorists to call the product "Windows for Warehouses."

This is really where our historical diversion began. Although it has taken Microsoft a while to join the advocates of peer-to-peer networking, it appears that the peer-to-peer model provides the direction for the company's own networking products in the foreseeable future—a direction reinforced by the release of Windows 95.

Microsoft's move to a reliance on peer-to-peer networking is hardly unique. Recent developments in distributed systems technology have begun to find their way into commercially available products, with remote procedure call capabilities and distributed object management features[6] moving from the realm of computer science research to production systems. Distributed systems tend to rely on the availability of an underlying peer-to-peer network architecture, and despite what Novell might say, client-server networking seems destined to become not much more than a network configuration issue over the near term.

Of course, the major improvements in Windows networking also allow Microsoft to prevent Novell from establishing any market share in desktop systems. Sure, you may continue to buy Novell servers, but the capabilities of Windows 95 make Microsoft your most likely desktop operating system supplier.

Networking Goals

Microsoft emphasizes the support for multiple network connections over the other goals for networking in Windows 95. You'll hear the term "universal client" used to characterize this particular goal. Here's what the term actually means:

- A set of architected interfaces that enable a network vendor to incorporate proprietary network client support into Windows 95.
- System support for simultaneous operation of a single Windows 95 system on several networks.
- A common user interface for network browsing, resource connection, and printing—regardless of the underlying physical network type.

6. Capabilities that Microsoft has already announced as an important part of its Cairo development project.

- Support for network operations from within the system shell. No longer is networking an "add on" component; it's a fundamental part of the system.

Acknowledging the entrenched position of both Novell NetWare and the UNIX-dominated TCP/IP networks, Microsoft has developed Windows 95 client support for both. Of course, Microsoft would like its own network solutions to become as popular as those of Novell, so Windows 95 has to be a good family member and support connections to Windows NT systems as well as existing Windows for Workgroups networks. Incorporating a peer server with good file and printer sharing capabilities allows Windows 95 to act as a capable, self-contained networking product.

Microsoft chose to develop its own client services for NetWare for Windows 95. This decision was largely a response to Novell's poor track record when it came to providing timely, high-performance client software for Microsoft operating systems. Early tests of Microsoft's client services for NetWare (reported in May 1994) showed some impressive results, with two to three times the performance of the Novell solution for Windows 3.11.

The other major goal for Windows 95 networking was to develop new 32-bit protected mode software for all the network components. Networking is a big winner when it escapes the limitations of real mode, the advantages corresponding to those that were gained by the introduction of a 32-bit protected mode filesystem. Overall performance improves, large software components such as network transports disappear from low memory, and the use of Windows 95's multithreaded architecture gives improved response and network throughput. Naturally, the network team had to obey the laws of compatibility, and Windows 95 still allows the use of older MS-DOS and Windows 3.1 network drivers.

Network Software Architecture

Like the new filesystem architecture, network support in Windows 95 relies on a layered design that separates functionality into several distinct modules. Early formalized approaches to network software design were among the first instances of this technique, and proponents of

existing network architectures, such as the OSI model, tend to be quite doctrinaire about the layered approach. As with most aspects of Windows design, though, implementation performance and memory requirements are paramount considerations. Although the designers of Windows 95 networking adopted a layered approach, practical considerations dictated a few design impurities. Figure 9-1 shows the overall network software configuration in a "typical" Windows 95 system that provides access to two networks through a single network adapter.

Many of the component names in Figure 9-1 are probably already familiar to you. We'll look at each of them as we analyze the architecture. Windows 95 networking is one of the best examples of the use of Microsoft's *Windows Open Services Architecture* (*WOSA*), and coming to grips with the networking subsystem is easier if you understand WOSA to begin with.

WOSA

Microsoft came up with the unwieldy WOSA name as an umbrella for a set of software components that, although originating in different projects, exhibited many similar characteristics. Much of the design impetus for WOSA came from the need for applications to interface to different networks, although WOSA can be applied to non-networked environments as well. Essentially, WOSA encompasses a series of interfaces designed to allow multiple software components with similar functionality to co-exist in the operating system. The user's interaction with an application ultimately results in the application's using the system's defined APIs to manipulate data. WOSA introduces the *service provider interface,* or *SPI,* that allows the OS to call system components (called *service providers*) to complete the processing of the data. Whereas the API is independent of the underlying hardware or service, the SPI remains hardware independent but is usually service dependent, and the service provider component itself is intimately connected to its target environment. As far as the user or an application is concerned, a service provider is simply part of the operating system. Figure 9-2 on page 350 illustrates the common components you'll find whenever WOSA is used as the system model. The standard configuration includes the API layer, the *routing* module, the SPI layer, and the underlying service providers. To get its work done, a service provider may call on any operating system functions or use other, lower-level, service providers (again by means of a defined SPI).

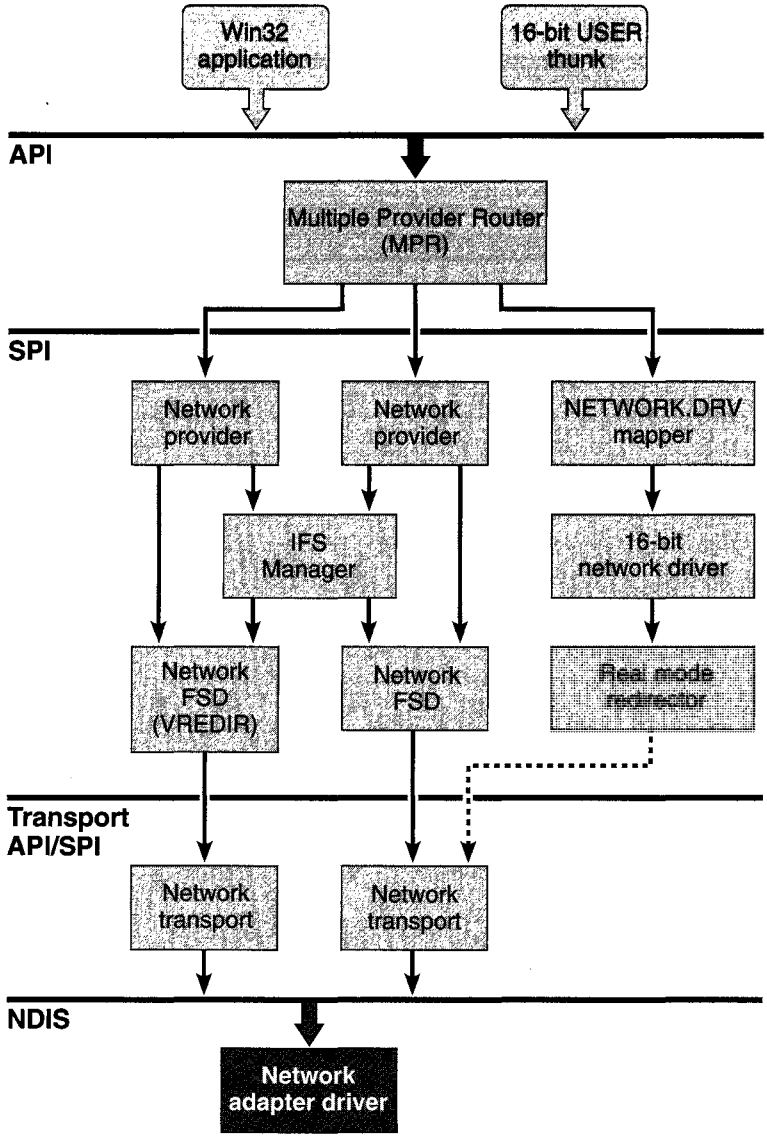


Figure 9-1.
Networking software components in Windows 95.

One good example of the use of WOSA is in an electronic mail application. Most heavy e-mail users today still have to learn at least a couple of different message editors, different mail addressing schemes,

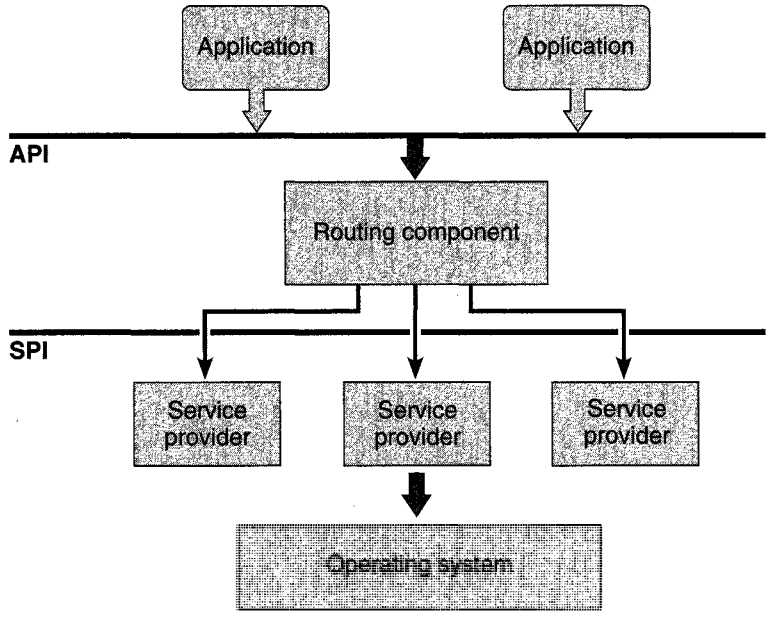


Figure 9-2.
Components in a standard WOSA configuration.

and idiosyncrasies of the underlying mail system. The desirable situation would be to prepare messages using a single application and have the underlying software figure out how to deliver the message—regardless of whether it's to someone in your office, to a CompuServe subscriber, or to a user out on the Internet. There are applications that try to do this, but from the point of view of the application developer, it's a daunting prospect to have to write a single application that knows everything about every electronic mail system. If you write the world's best message editor, you'd like to be able to hand a completed message to the world's best Internet mail delivery program, or to the world's best CompuServe mail delivery program, and so forth. Lower down in the system, the mail delivery programs themselves should have the option of using one of many different network transports to complete the physical transmission of data—and writing network transports is not what an electronic mail application vendor wants to spend resources on.

WOSA is the basis for providing this functional separation within Windows. In an extension of the example we've been considering, a mail message editor would use the Windows API. A mail service provider would implement the appropriate SPI (in this case, Microsoft's

MAPI), and Windows itself would link the components using the routing module. A similar arrangement would exist for other services. Several examples of the WOSA model already exist: the TAPI interface for telephone equipment manufacturers, the WinSock interface that standardizes the TCP/IP socket interface under Windows, ODBC for database access, and others.[7]

Network Layers

Looking back at Figure 9-1 on page 349, you can see the influence WOSA has on the Windows 95 networking subsystem. Networking support in Windows 3.1 was restricted to a single network. Windows for Workgroups expanded this to provide support for its native peer networking plus one other network. Windows 95 makes use of WOSA design techniques to allow you to install support for as many concurrent network connections as you want.[8] *The multiple provider router (MPR)* shown in Figure 9-1 is the routing component for Windows 95 networking. Both the *network provider* modules and the *network transports* conform to SPI rules, and at the lowest level, the popular *NDIS (Network Driver Interface Specification)* interface provides further support for shared device access and abstraction of the network hardware.

Here's a summary of the functions of each of the components illustrated in Figure 9-1:

API. The API layer is the standard Win32 API. Apart from file-based operations such as file open that happen to address remote filesystems, the Win32 API provides specific network-oriented APIs. These functions allow for such operations as remote resource interrogation and remote printer management. The *WNetGetUser()* API, for example, allows an application to determine the user name associated with a particular network connection. All Win32 network APIs have the *WNet* prefix.

Multiple Provider Router. The MPR is the routing component for Windows 95 network operations. The MPR also implements network operations common to all network types. The MPR

7. Each of these interfaces is a service provider. As you can see, marketing requirements dictate that an SPI must also have its own acronym.

8. An arbitrary implementation limit of ten networks was used in early releases of Windows 95. We'll have to wait and see whether ten equals infinity.

handles all Win32 network APIs, some of which may be routed to the appropriate network provider module. The MPR and the network provider modules are 32-bit protected mode DLLs.

Network provider. The NP implements the defined network service provider interface, encompassing such operations as making and breaking network connections and returning network status information. Only the MPR calls the network provider; an application never directly calls an NP.

IFS Manager. The IFS Manager fulfills its normal role of routing filesystem requests to the appropriate filesystem driver (FSD). The MPR won't see pathname-based or handle-based application calls; it's up to the IFS manager to route such calls to the network FSD. Network providers can call the IFS manager directly to perform file operations.

Network Filesystem Driver. Each network FSD is responsible for implementing the semantics of a particular remote filesystem. The FSD may be called by the IFS manager with requests of the same type as for local filesystems (for example, file open or file read), or the NP may call the network FSD directly. Obviously, a network vendor has to develop the NP and the network FSD together since each understands something of the semantics of the underlying filesystem, so these modules aren't interchangeable with others at the same level. Each network FSD is a 32-bit protected mode VxD. (This alone guarantees a substantial performance boost for Windows 95 networking.)

Network transport. The network transport VxD implements the device-specific network transport protocol. Windows 95 allows multiple transports to be in use simultaneously. The network FSD calls upon the transport for the actual delivery and receipt of network data. Given the likely network configurations of Windows 95 systems, each network FSD will probably use a particular transport. However, the separation of functions means that it's perfectly feasible for more than one FSD to use the same transport. Microsoft's NetBEUI and Novell's IPX/SPX are examples of network transports due to be delivered with Windows 95.

NDIS. The Network Driver Interface Specification is a vendor-independent software specification that defines the interaction

between any network transport and the underlying device driver. NDIS was originally developed to allow more than one transport to use the same physical network adapter and its associated device driver. NDIS has been revised over time, and Windows 95 networking supports NDIS version 3.0, although Windows 95 also contains provisions for using older 16-bit drivers conforming to either the ODI (Novell's *Open Datalink Interface*) model or earlier versions of NDIS. Both Windows NT and Windows 95 support the NDIS 3.0 interface, which means that network device driver developers only need to follow the appropriate rules to produce a single driver that works under either operating system.

Network adapter driver. The network adapter driver VxD controls the physical network hardware. The NDIS interface allows the driver to remain unconcerned about most network protocol issues—the driver simply works in concert with the network transports to send and receive data packets. Drivers designed for Microsoft's networking products are called *media access control,* or simply *MAC,* drivers. The driver does have to incorporate support for the Plug and Play subsystem in order to participate fully in the Windows 95 environment.[9]

Network Operations

Before we delve into the details of some of the Windows 95 networking software components, let's look at a few of the basic network operations Windows 95 supports and at some of the terminology that pervades Windows 95 networking. The screen in Figure 9-3 on page 355 shows a typical networking action—using the shell to wander around the network looking for something. Such wandering is called *browsing,* and the objects of the user's attention are various types of network *resources.* Here are the terms you'll see as you deal with this type of user action or in descriptions of the software that implements such an action:

A **resource** is a network object available for shared access—usually a printer, a collection of files grouped in a disk directory, or a communications device such as a fax or a modem.

9. The network adapter driver supports Plug and Play in concert with the NDIS.386 VxD, which is a standard component of Windows 95.

To **browse** is to wander the network looking for resources. The Windows 95 shell's manifestation of browsing is a series of windows that open to display successive levels of network resources.

To **enumerate** is to list or examine a set of related objects. A server may be sent a command requesting it to enumerate all of its resources, for example. The local shell would then display this list to the user during a browse operation.

A **connection** is a logical link between a local name, such as COM1:, and a network resource. Establishing and maintaining network connections is a principal function of the higher layers of the network subsystem.

A **domain** in Microsoft's networking architecture is a collection of servers and resources. Such a logical grouping allows for easier administration since a user's access privileges to the domain define the user's access to each server. A friendlier grouping concept, the Network Neighborhood, was introduced into Windows 95 early in 1994. Whereas a domain has a formal specification, the neighborhood is simply the network resources you choose to include there.

A **container** is an object that holds other objects. A domain, for example, acts as a container for network servers. Using container objects when browsing a large network is easier for the user, who will at first see a probably small list of container objects rather than a very long list of individual servers.

A **share point** is a disk resource that a remote user can connect to. All directories and files in the share point's subtree become part of the connected network resource.

The connection is particularly significant in Windows 95 networking. A network connection is essentially the ability to have references to the local LPT1: device be replaced with operations on a network printer \\Server1\LaserJetIII or a network file \\Server2\letters\letter.doc take the place of an apparently local file H:LETTER.DOC. Windows 95 formalizes the notion of a *persistent connection,* a network connection that has a lifetime beyond a single session or working day. You'll see persistent connections in use whenever you log in to the network. The shell remembers the connections that were in place the last time you logged

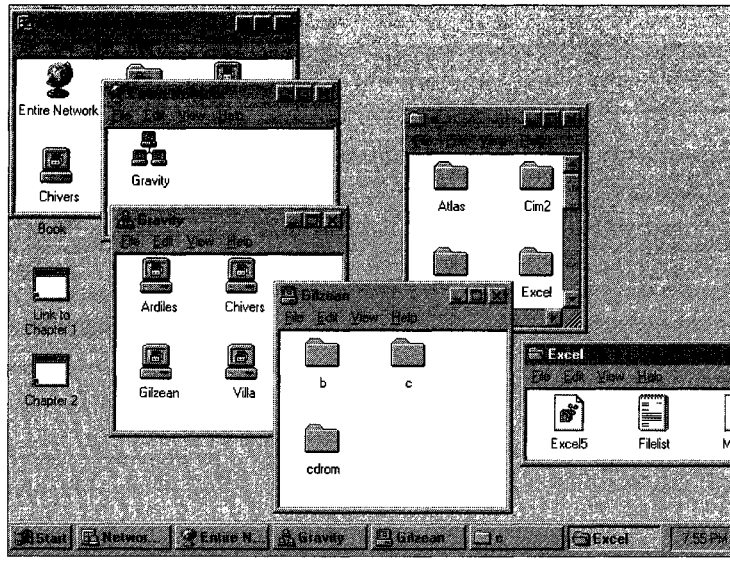


Figure 9-3.
Browsing the network with the Windows 95 shell.

in and restores them. If you use the same network printers and the same network mailbox each day, as most people do, you don't need to explicitly restore the connections every day. Windows 95 networking allows an application to identify a connection to a network resource as a persistent connection, and thereafter the shell will take care of restoring the connection—neither the application nor the network provider needs to worry further about having to set up the connection for each new session.

The Multiple Provider Router

Windows 95 provides the multiple provider router as a standard DLL. Functions within the MPR relieve each network vendor of the need to implement a large amount of common code. Equally as important, the fact that each NP relies on the same code in the MPR means that there will be a consistent treatment of many network issues. The MPR recognizes the fact, for example, that the names LPT1 and LPT1: refer to the same local device. Leaving such details up to each NP would almost guarantee some set of minor differences that would have the potential to confuse the user.

An application (including the system shell) is the principal cause of most MPR service calls. The MPR DLL resolves all the networking APIs defined for the Win32 interface. Microsoft refers to this subset of the Win32 APIs as "WinNet" or "WNet" functions, and every API in the subset uses *WNet* as a name prefix. To avoid any confusion, the functions provided by each network provider use *NP* as a name prefix. Application calls to WinNet functions may well result in the MPR's calling NP services, but applications never call the network providers directly.

The 32-bit WNet API functions are another example of the Windows 95 team's efforts to take advantage of the switch to 32-bit interfaces to improve on the API design. Apart from improvements in the network subsystem proper, enhancements in the Windows 95 base operating system add a lot to the Windows networking capabilities. Changes in the API reflect these improvements in Windows 95:

- Plug and Play technology is a major aid in reducing the complexity of setting up a network. The original release of Windows for Workgroups actually pioneered several aspects of the hardware recognition and configuration capabilities now incorporated in the Plug and Play subsystem.
- Support in the base system for long filenames was previously part of the network subsystem to allow interoperation with Windows NT and OS/2 LAN Manager servers, both of which support long filenames on certain filesystem types.
- Multiple concurrent network support obviates the need for some APIs.
- Common interfaces with Windows NT reduce both the application developer's and the device driver developer's workloads as they try to support both operating systems.

A number of Windows 3.1 APIs, though still supported for 16-bit application compatibility, have disappeared from the Win32 API set and have been declared "obsolete" by Microsoft. All the *LFN* prefix APIs that dealt explicitly with long filenames, for instance, are "obsolete."

Reducing the number of explicit network APIs obviously benefits the application developer, who now has less to learn when incorporating networking capabilities. The API reduction doesn't mean less functionality, however, since improvements in the base operating system also boost the networking capabilities of the average application. For

example, using UNC pathnames that reference network locations such as \\Server\Resource\Document_File is now recommended practice for every application. The filesystem supports this naming convention directly (through the *CreateFile()* API), and using full network pathnames is now just plain good programming practice rather than a convention limited to network-aware applications. The new filesystem architecture results in an API call that needs network services being routed to the appropriate network component. The application doesn't need to worry about calling a network-specific API.

32-Bit Networking APIs

Before we look at the services that must be supplied by a network provider, let's look at the APIs that are specific to a network environment. The Win32 network APIs fall into two main sets: the set of functions that deal with network connections, and a set of miscellaneous services that support other network features. Apart from applications' calling these APIs directly, network providers also call these APIs to take advantage of the common code implemented in the MPR.

Network Resources

Several of the WNet APIs use a data structure identified as a NETRESOURCE. This object is central to the interaction of the application and the underlying system and describes the type of the resource in addition to linking the resource to the underlying network provider that supports it. Figure 9-4 shows the NETRESOURCE data structure. Specific API calls may not use all of the fields in the structure, and in some cases, there is a *don't care* or *all* value for a field.

```
typedef struct _NETRESOURCE {
    DWORD  dwScope;
    DWORD  dwType;
    DWORD  dwDisplayType;
    DWORD  dwUsage;
    LPTSTR lpLocalName;
    LPTSTR lpRemoteName;
    LPTSTR lpComment;
    LPTSTR lpProvider;
} NETRESOURCE;
```

Figure 9-4.
The NETRESOURCE data structure.

If you examine the purposes of the fields in the NETRESOURCE data structure, you can begin to see the relationship between the application (particularly the shell) and the underlying network subsystem:

The *dwScope* field, when used in an enumeration function, specifies the scope of the enumeration. The scope can be all resources on the network, currently connected resources, or persistent connections.

The *dwType* field determines whether the resource type is a disk, a printer, or another type.

The *dwDisplayType* field identifies the resource as a network domain, a network server, or a share point for purposes of graphically displaying the network resource.

The *dwUsage* field denotes the resource as one that you can directly connect to or as a container resource.

The *lpLocalName* field points to a string that names the local device.

The *lpRemoteName* field points to a string that names the network resource.

The *lpComment* field points to a string that contains a comment supplied by the associated network provider.

The *lpProvider* field points to a string that contains the name of the network provider associated with the resource. (A NULL value indicates that the name of the provider is unknown.)

Connection APIs

The connection APIs allow applications to create and break access to explicit network resources. The connection APIs appeared in earlier versions of Windows networking, but the latest form of these APIs alters the format of the call parameters slightly, and although older APIs such as *WNetAddConnection()* are still supported, the recommendation is to use the most recent form (in this case, *WNetAddConnection2()*). Here's a summary of the connection APIs:

API Name	Function
WNetAddConnection()	Connect to a network resource using a local device name. Replaced by *WNetAddConnection2()*.
WNetAddConnection2()	Connect to a network resource using a local device name.
WNetCancelConnection()	Break an existing network connection. Replaced by *WNetCancelConnection2()*.
WNetCancelConnection2()	Break an existing network connection.
WNetGetConnection()	Retrieve the network resource name associated with a local device name.
WNetNotifyRegister()	Register a connection notification function.
WNetConnectionDialog()	Start a network connection dialog box.
WNetDisconnectDialog()	Start a network disconnection dialog box.

The connection APIs generally deal with NETRESOURCE structures—passing a structure with the fields necessary to complete the operation filled in. An application can call the *WNetConnectionDialog()* and *WNetDisconnectDialog()* functions directly to allow the user to make or break a network connection. These two functions are the same ones used by the shell for network browsing.

The services of a network provider are called on to help complete the connect or disconnect operation, but the NP doesn't need to be directly involved in the details of network browsing, resource selection, and persistent connections. However, the *WNetNotifyRegister()* API does allow the NP to watch network connections if it wishes. Using this API, an NP can register a callback that occurs before and after each network resource connect and disconnect operation initiated by the MPR. Within the callback, an NP can affect the operation in progress. For example, if a connect operation fails, the NP can use the notification callback to instruct the MPR to retry the connection attempt.

Enumeration APIs

The three enumeration APIs—*WNetOpenEnum()*, *WNetEnumResource()*, and *WNetCloseEnum()*—allow a caller to examine the details of the available network resources. You use these APIs much as you might use an MS-DOS FindFirst/FindNext sequence to search for a file on a disk. The *WNetOpenEnum()* API allows the caller to describe the set of target network resources, and successive calls to the *WNetEnumResource()* API will

return NETRESOURCE structures filled in with the details of the matching available network resources. The MPR will involve the NPs in completing the enumeration process, but the Win32 APIs cloak the details of a particular NP's enumeration functions. The user sees the result of a network enumeration as a series of open windows displaying the successive layers of the enumeration, as in Figure 9-3 back on page 355.

Error Reporting APIs

The *WNetSetLastError()* and *WNetGetLastError()* APIs are equivalent to the Win32 *SetLastError()* and *GetLastError()* functions normally used by DLLs. These functions allow a caller to set a specific error code that will be returned to another caller or to retrieve an extended error code. The network versions of the functions are provided for use by a network provider only and not as a general application interface.

Local Device Name APIs

The local device name APIs help an NP to manipulate device names consistently. Again, these APIs are intended for use by NPs only and are not for general application use. The *WNetDeviceGetNumber()* API will accept a device name string and return a local device number—the MPR carries out all the necessary name validation and matching during the call. The *WNetDeviceGetString()* function reverses the procedure, returning a name for a given device number. The *WNetGetFreeDevice()* function simply returns a currently unused local device number.

UNC APIs

The UNC APIs are designed to provide a service to the network providers that allows consistent treatment of UNC pathnames. For example, MS-DOS naming conventions call for the \ character as a pathname component separator, whereas a UNIX system uses the / character. UNC naming support is available for both environments, however. The *WNetUNCValidate()* API function checks a complete pathname, and the *WNetUNCGetItem()* API returns successive components of the name to the caller.

Password Cache API

Windows 95 networking implements a local password cache scheme that encrypts passwords and stores them locally. The administrator can disable this scheme (for extra security), and an NP can prevent its passwords from being retained in persistent storage. *WNetCachePassword()* is the API that provides access to the password cache services.

Authentication Dialog API

The *WNetAuthenticationDialog()* API provides a service that allows an NP to request authentication information—particularly a user name and password—from the user. Again, the intent is to present a consistent network access interface to the user, regardless of the underlying network type.

Interfacing to the Network Provider

The MPR is responsible for loading each NP in turn. The settings in the Windows SYSTEM.INI file determine the total network configuration for a particular machine. Figure 9-5 shows a section of a SYSTEM.INI file that describes a three-network configuration—Windows for Workgroups, NetWare, and the revolutionary NewNet product.[10] The loading and initialization order for network providers will be the order in which they're specified in the SYSTEM.INI file. Each NP can store additional initialization information within its private section of the SYSTEM.INI file, but values for the *NPID*, *NPName*, *NPDescription*, and *NPProvider* fields are required, and Microsoft has reserved all strings with the *NP* prefix for its own use. The *WNetGetSectionName()* API allows an NP to find its private section within the SYSTEM.INI file.

```
[BOOT]
Networks=WFWG, NetWare, NewNet
[WFWG]
NPID=0x0002
NPName=Windows
NPDescription=Microsoft Windows Network version 95
NPProvider=wfwnet.drv
[NetWare]
NPID=0x0003
NPName=Novell NetWare
NPDescription=Novell NetWare version 3.11
NPProvider=netware.drv
[NewNet]
...
```

Figure 9-5.
SYSTEM.INI entries for multiple (three) networks.

10. The latter product is unlikely ever to see the light of day but is useful for illustrative purposes.

The *NPProvider* field identifies the DLL that implements the network provider interface. The *NPID* field identifies the type of the network. Figure 9-6 shows a partial list of the network products identified for support—which says something for how serious Microsoft is in its intention to allow a Windows 95 system to connect to just about anything you can put on the other end of the wire. Simply adding the name of an existing network driver to the SYSTEM.INI list doesn't magically get you network support, though: the DLL that provides the network interface must be a full Windows 95–compatible network provider, and it's up to the various vendors to produce this software themselves.[11]

Mnemonic Identifier	Supported Network Type
WNNC_NET_MSNET	Microsoft MS Net
WNNC_NET_LANMAN	Microsoft LAN Manager
WNNC_NET_NETWARE	Novell NetWare
WNNC_NET_VINES	Banyan VINES
WNNC_NET_10NET	TCS 10Net
WNNC_NET_SUN_PC_NFS	Sun Microsystems PC NFS
WNNC_NET_LANTASTIC	Artisoft LANtastic
WNNC_NET_AS400	IBM AS/400 Network
WNNC_NET_FTP_NFS	FTP Software NFS
WNNC_NET_PATHWORKS	DEC Pathworks
WNNC_NET_POWERLAN	Performance Technology PowerLAN

Figure 9-6.
Some of the network types supported in Windows 95.

The Network Provider

A single network provider implements the service provider interface for a particular network as a Windows DLL. The NP doesn't have to worry about multiple network issues or about most aspects of interfacing to the user. The MPR and the support that comes from the underlying filesystem architecture take care of all this. In fact, Microsoft's design recommendations for network vendors specifically deter the

11. By shipment time, this list may well have changed—not least because some network vendors may no longer exist.

implementer from using private user interface dialogs. This isn't to say that the characteristics of a particular network are totally hidden from the user. In several instances, the NP can register functions that the MPR will call—to extend its default handling of network browsing operations, for example.

The MPR will load the NP if its associated network is listed in the SYSTEM.INI file as active. Since the NP is a Windows DLL, the system will call its standard initialization entry point once the NP is loaded. This allows the NP to carry out any private initialization it needs to. Thereafter, the NP responds to the MPR by means of the defined network provider interface. Many of the defined NP functions are optional—the NP supports them only if it has something to add to the default actions of the MPR. For example, the NP doesn't need to implement the group of functions responsible for enhancing the graphical display of network resources unless it wants to alter the shell's representation of the resources. The MPR also has to determine what the NP can support—for example, whether the NP is able to handle UNC pathnames completely.

To figure out exactly what the behavior of a particular NP is going to be, the MPR calls the *NPGetCaps()* interface. The parameter to this call is a query about a particular NP capability or about an NP characteristic (the supported network type, for example). In the case of a query about a capability, the response from the NP determines whether the MPR will subsequently call the specific interfaces that implement the feature or rely on its own default handling. NPs don't need to implement stub routines or return errors for unsupported interfaces—once the MPR recognizes that an NP doesn't support a particular capability, it won't try to call any of the related interfaces.

There are also times when the MPR calls each NP in turn, trying to find an NP that recognizes a particular resource. An error return from one NP causes the MPR to move to the next, finally returning an error to the caller if no NP responds successfully.

Network Provider Services

Let's take a look at the details of the service provider interface for an NP. Apart from the *NPGetCaps()* interface just described, there are six groups of functions:

User identification. The single *NPGetUser()* interface that allows the caller to determine the current username associated with a particular network resource.

Device redirection. The interfaces that make, break, and manipulate network connections.

Shell interface. Functions that augment the native display behavior of the shell during browsing and other operations.

Enumeration. Functions that an NP must support if it supports browsing operations.

Authentication. Functions that support the network-specific security features.

Configuration. Two optional interfaces: *NPEndSession()*, to notify the NP that Windows is closing down, and *NPDeviceMode()*, to allow network-specific configuration actions, such as choosing a network adapter from among those available.

All of the functions share similar calling and error return conventions.

Device Redirection SPI

The device redirection set of NP interfaces is the eventual target of the WNet connection APIs that form the associations between drive letters (A: through Z:) or device names (LPT1: and so on) and network resources. Some networks don't need local devices for network connections—a characteristic that a network reports through the *NPGetCaps()* interface. The optional *NPValidLocalDevice()* interface allows an NP to restrict the set of local devices that the MPR can use to make connections through the NP. For example, the NP may support only LPT1: and LPT2:, whereas Windows 95 supports additional LPT devices. If the NP doesn't export the *NPValidLocalDevice()* function, that's an indication that the NP can handle any local device name.

NPNotifyAddConnection() is the callback function an NP can use to involve itself more directly with the network connection process. Here's the set of functions it belongs to:

NPAddConnection()	Make a network connection.
NPCancelConnection()	Break a network connection.
NPGetConnection()	Obtain information about a connection.
NPNotifyAddConnection()	Arrange a callback during network resource connection and disconnection.
NPValidLocalDevice()	Indicate whether a local device is valid for use as a network connection (optional).

Shell SPI

The shell interface functions assist the shell in displaying the network layout and the attached resources for the user. Several of these functions are optional. If an NP is happy with the default displays generated by the shell, it doesn't have to support the possible extensions. Here's a summary of the shell NP functions:

NPGetDirectoryType()	Provide information about a network directory.
NPSearchDialog()	Assist in network browsing.
NPFormatNetworkName()	Change the display appearance of a network pathname.
NPGetDisplayLayout()	Customize the appearance of the network layout.
NPDisplayCallback()	Call back during network display.
NPGetEnumText()	Return additional text information during display.
NPGetNetworkFileProperties()	Display file properties.
NPDirectoryNotify()	Notify of directory creation, deletion, and movement.

The *NPSearchDialog()* function extends the standard shell browsing mechanism, allowing an NP to display its own view of the associated network. If an NP supports this extension, the shell enables a Search button in its connection dialog. If the NP doesn't support the enumeration interfaces, the shell will use its private search facility exclusively for browsing.

Enumeration SPI

The enumeration functions are an all or nothing subset—if the NP responds to a query from the MPR by indicating that it supports enumeration, it must support all four functions. If an NP doesn't support network browsing, it doesn't need to implement the enumeration functions. Within an NP that supports them, the open, enumerate, and close functions are the eventual target of the corresponding WNet enumeration APIs. The *NPGetResourceParent()* SPI assists the shell in browse operations by providing a means of moving back up a hierarchy. The enumeration functions are shown on the next page.

NPOpenEnum()	Begin enumeration.
NPEnumResource()	Enumerate network resources.
NPCloseEnum()	End enumeration.
NPGetResourceParent()	Return the parent of a specified network resource.

Authentication SPI

The authentication functions allow the NP to participate in the network logon and logoff procedures controlled by the MPR. During the logon process (see Figure 9-7), the NP has the opportunity to carry out additional user authentication and to provide the MPR with the name of an executable file it can use as a logon script. The shell will restore the user's persistent connections for the network during the logon. Here are the authentication functions:

NPLogon()	Log on to the network.
NPLogoff()	Log off the network.
NPGetHomeDirectory()	Return the user's personal network directory.
NPChangePassword()	Notify of a successful change of the user's password.

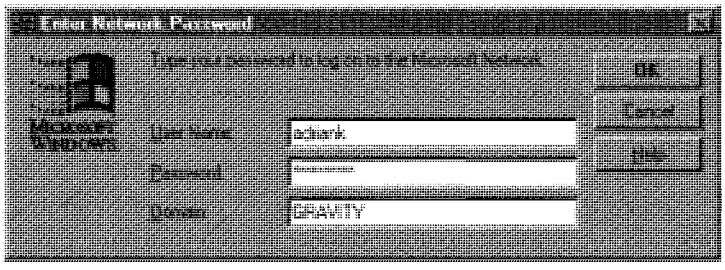


Figure 9-7.
Standard network logon dialog box.

Network Transports

Windows 95 has not revolutionized the world of network transports. Network transports still play the same role: they provide reliable, sequenced, error-free connections among the upper-level network software modules. Windows 95 also has to live within the constraints of

compatibility—particularly for existing real mode network device drivers—and the networking subsystem incorporates features that allow the continued use of these drivers by the transports.

The network transport has to play two basic roles within the system: it must act as a communications medium for the network FSDs as they provide support for the file and print services, and as an API usable directly by network applications. In both cases, it's the published transport protocol interface that comes into play. Windows 95 supports both NetBIOS (via Microsoft's NetBEUI transport) and Novell's IPX/SPX protocols. The transports for both protocols are full 32-bit protected mode modules supporting 32-bit and 16-bit application interfaces. These days, network applications such as client-server databases and network management systems tend to make use of higher-level network protocols (named pipes or Microsoft's ODBC, for example) rather than deal directly with the transport interface. But there are plenty of important applications still written to both the NetBIOS and the IPX/SPX interfaces.

In the medium term, Microsoft has begun to recommend use of the Windows Sockets interface for network applications. The project to define the so called WinSock interface was a multicompany attempt to rationalize all of the different versions of the TCP/IP[12] protocol–based socket interface that various vendors had ported to the Windows environment. Originally introduced as a networked interprocess communications mechanism with version 4.2 of the Berkeley UNIX system, the socket interface has become a popular API. Although the sockets lineage goes back to the TCP/IP world, sockets can be implemented on top of other transport protocols. The Windows Sockets project was so successful that, in addition to using Windows Sockets as an interface to the TCP/IP world, Microsoft developed a Windows Sockets module that uses NetBEUI as its underlying transport.[13]

In the longer term, the need for fully distributed applications will make an RPC-based method the preferred network application interface. Windows NT has already begun to emphasize the use of RPC interfaces, and Microsoft's Cairo system will underline their long-term

12. TCP/IP is now officially called the *Internet Protocol Suite.*

13. Windows 95 will include a TCP/IP transport and several related utilities such as FTP, Telnet, and Internet access programs.

importance. However, the migration from a simple client-server application model to a fully distributed one is not yet upon us, so the simpler network programming interfaces supported by Windows 95 will remain important for some time to come.

Network Device Drivers

Microsoft defines a *media access control,* or *MAC,* device driver model. A MAC driver is the lowest-level software in the networking subsystem and deals directly with the network adapter. A MAC driver conforms to the *Network Driver Interface Specification* (*NDIS*). So called clients of the MAC driver—the transport protocol modules—access the MAC driver functions via the NDIS interface (a process termed *binding*). The NDIS specification was originally developed for Microsoft's OS/2 LAN Manager product and has become fairly widely used on network systems that don't use a Microsoft OS. NDIS is now at version 3.0. The development of this most recent version of the specification was done largely by the Windows NT group.[14]

NDIS aims to provide solutions to a number of problems inherent in a complex network environment:

- Hardware independence. The interface between the transport protocol and the MAC driver ought to allow at least source code portability for the transport software.
- Transport protocol independence. The MAC driver has to be hardware dependent, but the NDIS interface ought to allow the use of the driver by any network transport.
- Multiple transport protocols. The interface to the driver needs to allow more than one protocol to share a single network adapter (and a single Ethernet cable).
- Multiple network adapters. NDIS has to allow the simultaneous use of more than one network adapter in the same host machine (possibly using a single MAC driver).

14. Along with other general improvements to the specification, Windows NT required that NDIS 3.0–compliant software be usable in a multiprocessor environment. The Windows 95 team didn't have to worry about this particular requirement.

- Performance. Network vendors strive continually to win benchmark competitions: if using NDIS implies poor performance, it's unlikely to be a very popular interface.[15]

You can think of NDIS as an interface that allows multiple transport protocols to talk to multiple network adapters, possibly on a multiprocessor machine. Despite their graduated degrees of freedom, NDIS-compliant drivers are not that difficult to develop, and any network adapter you buy will probably come with an NDIS driver. Of course, the adapter may not yet come with a protected mode NDIS version 3.0 driver—and that's a problem the Windows 95 networking team had to address directly.

Although the NDIS model has achieved wide acceptance, there's another company in the networking business that has a different way of doing things. Novell's *Open Datalink Interface* (*ODI*) specification mirrors Microsoft's NDIS in aiming to define a protocol-independent device interface. And there are a lot of ODI drivers available too. In addition to needing to provide compatibility for older NDIS drivers, Windows 95 had to support ODI drivers.

Network Driver Compatibility

To solve the problem of supporting non-NDIS 3.0 network device drivers—specifically NDIS 2.0 and ODI drivers—Microsoft has evolved a series of low-level modules, sometimes called *helper* modules, that act as "glue" between the various interfaces. This allows the Windows 95 protected mode NetBEUI transport to use an NDIS version 2.0–compliant real mode adapter driver, for example, or a real mode IPX/SPX transport and associated ODI driver to operate alongside a NetBEUI configuration.

Essentially, the helper modules present an upper-level interface that complies with the caller's requirements, and they translate the calls to a lower-level interface that matches the capabilities of the available device driver. In some cases, the helper module may simply manage the transition between protected mode and real mode (actually virtual 8086 mode). You can recognize the type of the helper module as

15. NDIS is specified as a C language interface, and for performance reasons many of the NDIS function calls are implemented as inline code using macros.

either a protected mode VxD (with a .386 filename suffix) or an MS-DOS TSR (with a .SYS filename suffix). The PROTOCOL.INI file is set up to contain the description of how all the pieces fit together in a running system.

Network Configurations

Putting together the jigsaw of network transports, drivers, and compatibility helper modules yields some interesting configuration possibilities. Figure 9-8 illustrates the simplest case—a single network adapter with a protected mode NDIS 3.0–compliant driver. The additional module illustrated—the VNETBIOS component—virtualizes the access to the transport for the concurrently running virtual machines.

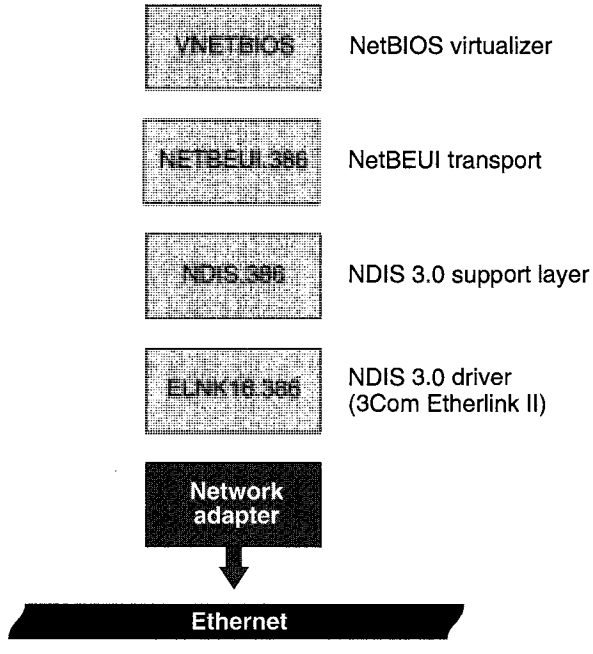


Figure 9-8.
A simple NDIS 3.0 network configuration.

Figure 9-9 illustrates a configuration that supports the NDIS 3.0–compliant NetBEUI transport running together with a real mode NetBEUI transport. At the lowest level, the network adapter driver is an NDIS 2.0 real mode driver (UBNEI.DOS in the example). The helper

modules NDIS2SUP.386 (a protected mode VxD) and NDISHLP.SYS (a real mode MS-DOS TSR) merge these different interfaces into a workable configuration.

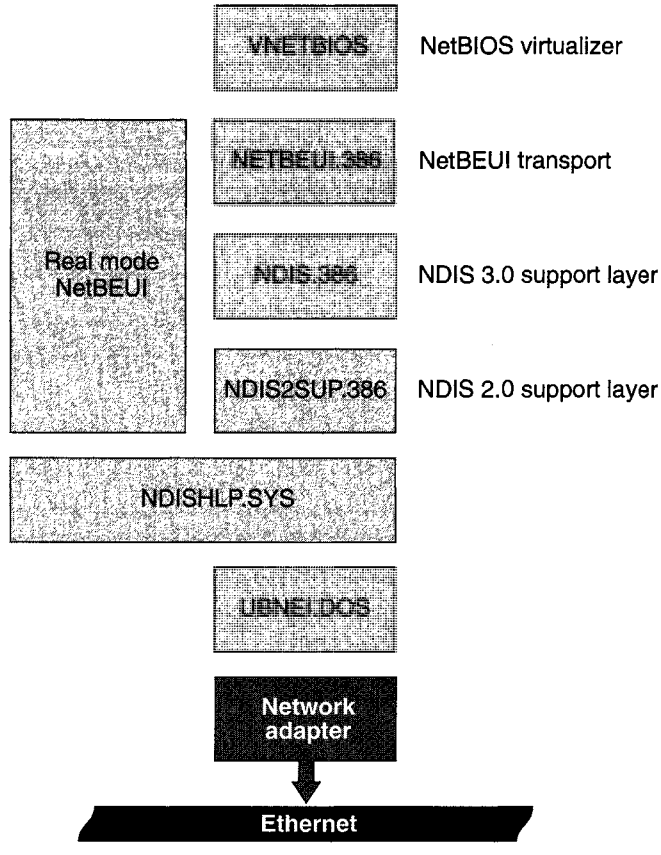


Figure 9-9.
Mixing NDIS 2.0 and NDIS 3.0 in a single network configuration.

Although it seems highly unlikely that the configuration illustrated in Figure 9-10 on the next page would have a life outside Microsoft's test labs, it does serve to show the extent of the compatibility provided under Windows 95. This configuration shows four separate transport protocols in use—Novell's IPX/SPX, the purely illustrative ABC protocol, and NetBEUI and TCP/IP cloaked by the Windows Sockets interface. The lower layers again use a combination of protected mode and real mode helper modules to form the paths to and from the network adapters.

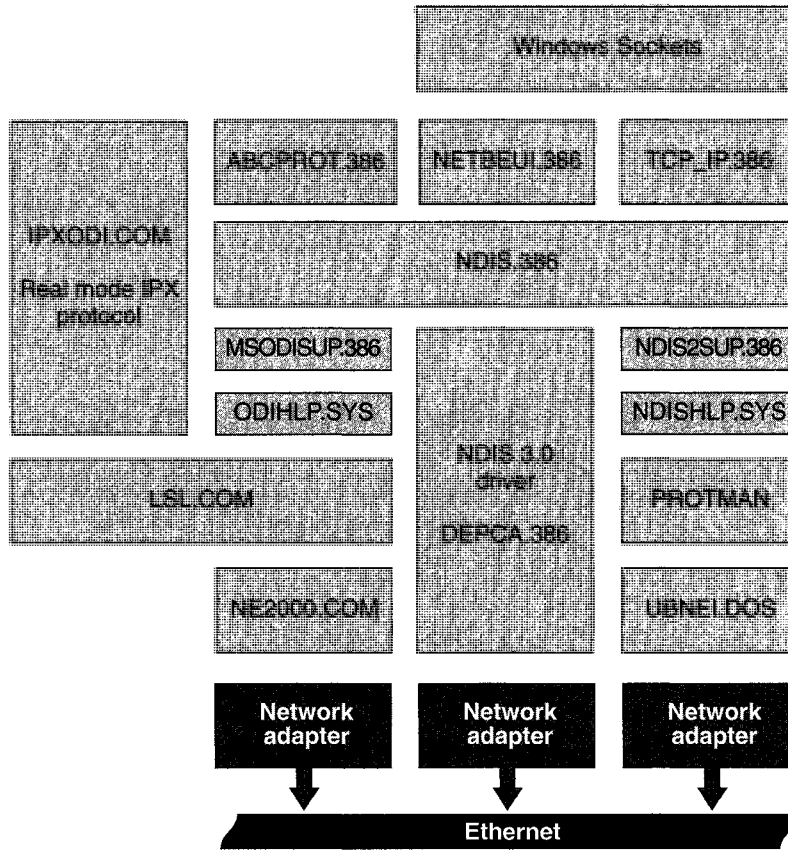


Figure 9-10.
A complex network configuration—multiple protocols and multiple adapters.

The Network Server

The peer-to-peer capability of Windows 95 networking means that there has to be a server available for use on the local machine. Although the Windows 95 networking group is not trying to compete with the high performance and industrial strength of Microsoft's own Windows NT Advanced Server product, they have produced a highly capable server with performance exceeding the levels reached in Windows for Workgroups version 3.11. As in previous versions, the server supports file and printer sharing features, giving you the option to provide other network users with access to files, directories, and printers

local to your machine. In response to many customers who want to prevent their users from running desktop systems as network servers, Windows 95 can be configured to run as a client machine only. Figure 9-11 on the next page shows how the Windows 95 server software interfaces with the other network components.

Server Components

The major server component is a ring zero VxD named VSERVER that provides the bulk of the local file and printer access capability. The server utilizes the defined installable filesystem interfaces for access to the real data on local hard disks and CD ROM devices and interacts with the print spooler to support the printer sharing feature. Here's a summary of what each component is responsible for:

Spooler. The print spooler exists at the application level (in ring three) and also as a system component (a VxD running at ring zero). There's a shared memory interface for communication between the ring zero and ring three components, and a ring zero API that allows the server to submit a print job to the ring zero spooler.

MSSHRUI. The Microsoft share point user interface component is a ring three DLL that the shell uses as it satisfies user-initiated operations such as adding new share points to the local machine.

VSERVER. The main server software component itself is multi-threaded, maintaining a pool of threads that it allocates among the different network requests. The server accesses the network directly using the transport level interface and accesses the local file systems through the IFS Manager.

Access Control. The Access Control VxD controls individual file access requests, using the provided username and filename to verify the rights of the particular user to access a shared resource.

Security Provider. The Security Provider component takes responsibility for authentication of network access requests. It uses the combination of the user's login name and supplied password to verify the legality of any access request.

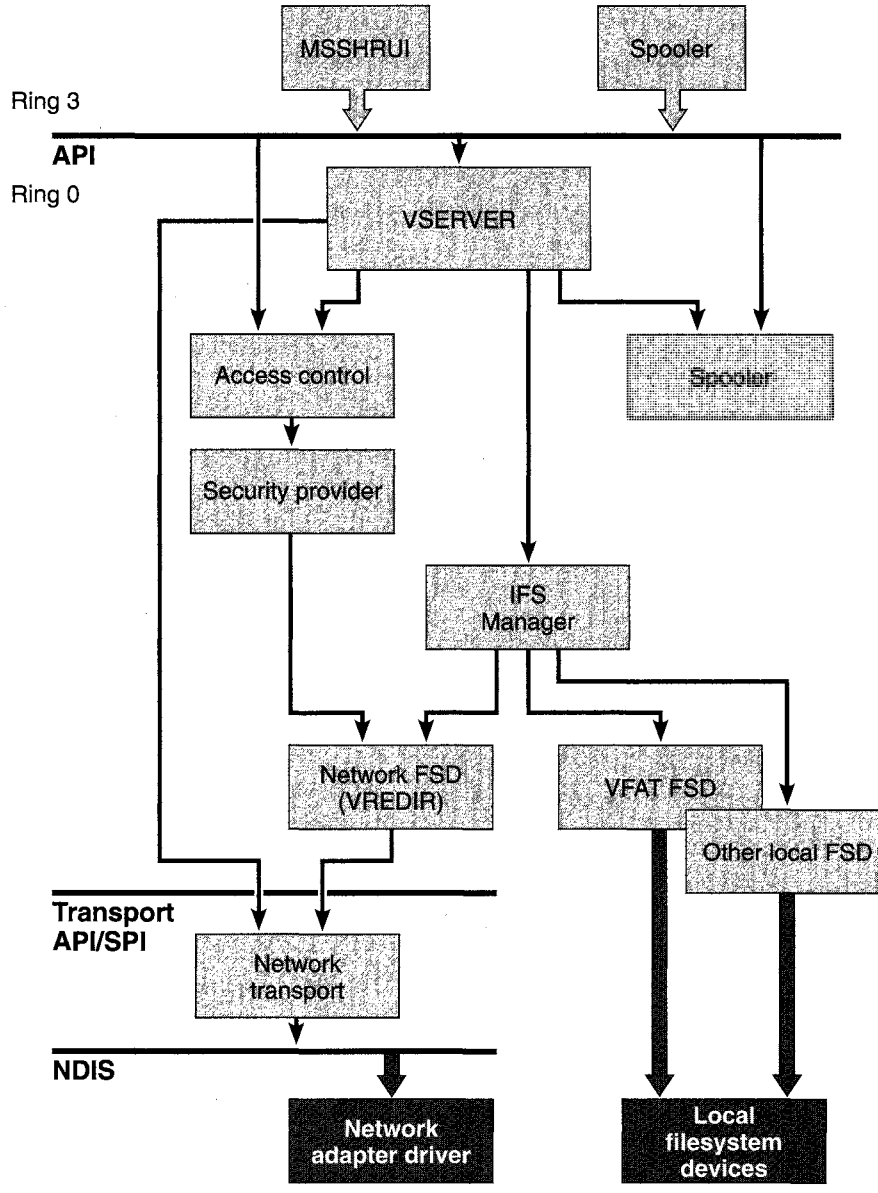


Figure 9-11.
Windows 95 Network Server architecture.

Network Printing

Of all the features of Windows, printing is perhaps the most commonly used and the most difficult for an average user to come to grips with. The complexity inherent in supporting hundreds of different types of printers—each with many configuration possibilities—and the layers of obfuscation added by a network can make printing under Windows 3.1 a painful experience. Even Microsoft's own Windows Printing System product fails to solve the network printing problem, although it does a good job of supporting a locally attached printer. Windows 95 aims to solve these problems with a new printing architecture whose design was borrowed from Windows NT and then adapted. Figure 9-12 on the next page illustrates the major components of the printing subsystem.

In common with the network file access capabilities, the printing system uses a routing component (the *Print Request Router*, or *PRR*) that accepts Win32 API calls and directs them to a *print provider* (*PP*). A single system may host several print providers if there are connections to multiple printers. The PP translates the information in the API call to a form suitable for the underlying network—for example, the printer might be attached to a NetWare server—and passes it on. The PP will convert the returned information to the correct Win32 format and pass it back to the application. The application itself doesn't need to know anything about the printer's capabilities or any network connection details. Although it will include several print providers as standard components, Microsoft's intent is that the printer manufacturers themselves will produce their own print providers. The printing architecture allows for multiple PPs related to a single printer to install themselves. So, for example, the generic PP for an HP LaserJet might be overridden by the better "quality of service" offered by a Hewlett-Packard–produced PP.

Locally attached printers participate in this printing architecture, with the local print provider interfacing to the resident printer driver and the spooling system. The printing architecture also allows for the inclusion of a *monitor* within the chain of modules that collaborate during the printing process. A monitor takes responsibility for low-level interaction with the printer. In the case of a printer attached to a bidirectional port, the monitor enables intelligent error handling and

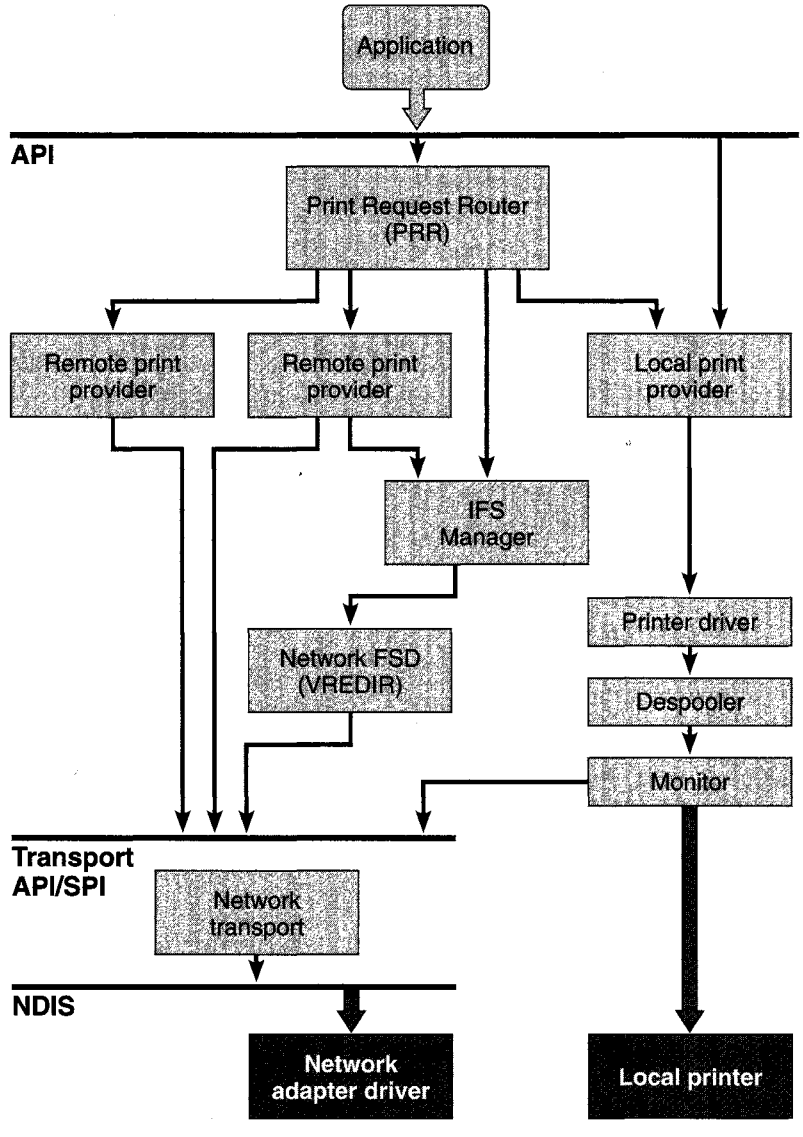


Figure 9-12.
Windows 95 Network printing architecture.

printer management.[16] More recent product innovations, such as printers with built-in network adapters that attach directly to the network, can also be handled by means of the monitor mechanism. The monitor simply talks to the printer via the network transport interface. The upper layers of software don't know and don't care about the specifics of the printer's connection.

One of the design goals of the Windows 95 team was encapsulated in the phrase "point and print," which was used during many early product presentations. What it meant was the ability of the user to print by simply dragging a document icon to a printer icon on the shell desktop and dropping it. Windows then figures out how to print the document, restoring a network connection if necessary and even loading the appropriate printer software dynamically. No longer does the user need to know a printer's exact model number designation and the amount of memory in the printer, which might be fifty yards away—let alone need to have a copy of the right Windows installation diskette handy. The point and print capability is supported by several new APIs that enable the shell to determine the available printers and associated drivers and then to dynamically load a printer driver.[17]

Network Security

Microsoft's emphasis in the design of Windows 95 networking security was on providing good security for the Windows 95 system itself and enabling a Windows 95 machine to participate in the security system implemented in a more complex scheme. The design of the FAT filesystem alone means that a Windows 95 machine is probably insecure—at least not up to the level of security required by the stringent government specifications that Windows NT complies with. In fact, presentations of the Windows 95 network security feature usually include some form of this statement: "if you want something that's small, fast, and easy to use, we have it; if you want something that's bulletproof,

16. Microsoft's Windows Printing System was the first product to make use of this bi-directional capability within Windows. Although the Windows Printing System was a great product for locally attached printers, it didn't support network printing. The Windows 95 printing architecture fixes that problem.

17. If you're searching for details, the *EnumPrinters()*, *GetPrinter()*, *GetPrinterDriver()*, *GetPrinterDriverDirectory()*, *GetPrintProcessorDirectory()*, and *LoadLibrary()* functions are those most intimately involved with the point and print capability.

use Windows NT." In the business world, most network administrators have to worry about some level of security protection and only a few have to concern themselves with protection against sophisticated break-ins. Windows 95 aims to meet the majority's need, and Windows NT is there for those who need a higher level of security.

Windows 95 provides two types of security:

- *Share-level security* similar to the security scheme in Windows for Workgroups. An administrator configures each network share point with a particular set of access rights.
- *User-level security.* A user's network name implicitly grants the user a defined set of access rights to each network resource.

Earlier designs for the system allowed for an additional security type—one that made use of a technique called *pass through authentication.* This technique would have allowed Windows 95 to pass a supplied login name and password to another system so that the other system could validate the user's security credentials and return access rights for the user to the Windows 95 host. The feature wasn't greeted with much enthusiasm, and it was dropped from the product. In the current design, a single system can operate under either share-level or user-level security—you can't mix the two types of security on one system. Most likely, every system in an organization will be set up with the same type of security.

Access Controls

A user's access to network resources is determined by what Microsoft calls *access controls,* also referred to simply as ACLs—for "access control lists." The ACL is the system data structure that describes access rights. In Windows 95, access controls can be applied to files, printers and a remote administration capability. Microsoft planned to incorporate security and other administrative functions together in a *System Policy Editor*—a utility aimed at supporting all of the network security and management features.[18]

18. This utility had appeared in various incarnations in Microsoft LAN Manager, Windows NT, and Windows for Workgroups. It was a late arrival in Windows 95. It wasn't folded into the product until after the Beta-1 release in June 1994.

Share-Level Security

Share-level security applies a set of permissions to an individual resource—regardless of which user is trying to gain access to the resource. The resource can be either a file (typically a subtree within the filesystem) or a printer. The administrator can protect a resource with a password that allows either full (read and write) access or read-only access. If a user knows the password, he or she has access to the resource.

User-Level Security

User-level security allows you to specify the names of individual users who have access to shared resources. For convenience, you can collect users into groups and give access permissions to an entire group—implying that every user belonging to the group gets the same access permission. To gain access to a resource, the user must belong to the set of users granted the appropriate permissions.

Conclusion

Windows networking has evolved from support for a single network with primitive setup facilities to a complete architecture supporting multiple network connections. The structure of Windows 95 networking relies heavily on Microsoft's WOSA design, and with support from the new installable filesystem interface, the networking architecture ought to be able to stand unchanged for several releases. As we'll see in the next chapter, the implementation of remote communications features is greatly simplified by the underlying support of Windows 95 for network components.

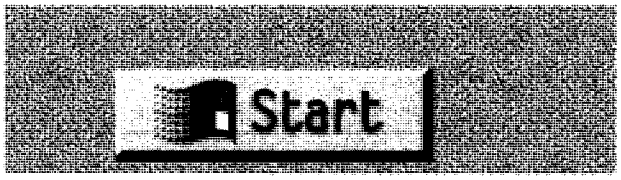
We haven't looked at a couple of features of Windows 95: the remote procedure call (RPC) capability and the collection of administrative features bundled together under the heading "systems management." The RPC facilities in Windows 95 are essentially identical to those available in Windows NT, and although Windows 95 itself doesn't make use of the RPC capability as extensively as Windows NT does, certain Windows 95 components, such as the network printing subsystem, do use RPC. The systems management features of Windows 95 incorporate all the administrative capabilities common to networked systems—assigning users to named groups, granting a user certain administrative privileges, and so on.

The new networking design allows any vendor to provide network access for Windows 95, although it's hard to see why a product that provides out of the box support for Microsoft, Novell, and TCP/IP networks would need to be augmented. Now that the operating system underlying the networking architecture is much more sophisticated, the peer-to-peer capability and overall performance ought to provide competition for the smaller networking companies. Although its security features don't match the rigorous approach taken by Windows NT, for many small to medium-size networks, Windows 95 will probably provide all the networking facilities that are needed. It will be interesting to observe the impact of Windows 95 on the local area networking market.

Sophisticated local area networks are at the upper end of the market Windows 95 addresses. The Windows 95 team also had a mandate to provide very good support for the other end of the market—for the ever-shrinking portable computer now used in a variety of "on the road" situations and for the burgeoning consumer market for multimedia applications. Those markets and Windows 95 support for them are the subjects of the next chapter.

Reference

Tanenbaum, Andrew. *Computer Networks.* 2d ed. Englewood Cliffs, N.J.: Prentice Hall, 1989. The standard tome on networking. If it isn't in this book, either it's not worth worrying about or it's fresh out of the research lab.



CHAPTER TEN

MOBILE COMPUTING

Many of the new features of Windows 95—the 32-bit operating system and 32-bit applications, the new rich visuals of the shell, and the built-in local area networking capabilities—call for the use of a fairly high powered desktop system. But the Windows 95 development team also had to address the needs of a large class of users who don't have continuous access to a powerful desktop computer. These users are loosely classified as "mobile," meaning that they use computers in various physical locations at various times. Some users are truly mobile—using only laptop computers and traveling frequently, retaining contact with their home bases or their customers via electronic mail, phone, and fax. Other users may move between only two locations—their offices and their homes—each location having a desktop system with somewhat different capabilities from the other's but the work at hand traveling back and forth and the work task remaining fundamentally the same.

Add to this already established need for mobility the recent market data that shows sales of portable computers growing more rapidly than sales of any other machine type, and sales of modems exceeding even wild expectations—and it's clear that Windows 95 needs to be a good product for smaller machines and for communications. Of course, the much vaunted era of the personal digital assistant (PDA) is now officially upon us too. Although from a practical standpoint the use of general purpose PDAs remains limited and frustration prone, Microsoft has invested considerable effort in the development of handwriting recognition technology and an integrated application, WinPad, targeted at PDAs.

In this chapter, we'll look at a collection of Windows 95 capabilities loosely grouped under the heading "mobile computing": communications support, electronic mail and fax support, and portable system

support. A lot of the communications support relies on features of Windows 95 that we've already examined: the layered network architecture and the WOSA service provider capabilities. And there are aspects of other features, such as Plug and Play, that take on even greater importance when smaller portable systems are involved. But to meet Microsoft's goals of great connectivity and what it sometimes refers to as "here, there, and everywhere computing," Windows 95 includes several new software components with important roles.

Remote Communications Support

The design of the communications subsystem in Windows 95 is derived largely from the design of the local area networking subsystem we looked at in Chapter Nine. An important aspect of the Windows 95 network software design is its ability to support many simultaneous connections via different network protocols and network transports. One or more of those connections can go from the user's machine via the communications subsystem to a remote network or to another communications provider such as a bulletin board system or an electronic mail gateway. From the user's perspective, the Windows 95 shell integrates access to remote systems with local area network access, and at least for file sharing and printer sharing purposes, remote communications looks and acts the same as any other network connection.

This consistency is maintained in applications written to make use of remote services: the Win32 API provides a consistent interface regardless of whether the needed resource is a file on the network server down the hallway or a file back at your main office thousands of miles away and accessible only by modem. Applications don't have to take special account of these different physical connectivity characteristics (although some optimization is possible if they do). Windows 95 provides all the glue necessary for the various system components to make each type of connection. And, naturally, for applications that will exploit characteristics of the remote connectivity features, many specific Win32 APIs offer that capability.

New in Windows 95 is the Windows Telephony API—TAPI for short. This new set of Win32 interfaces integrates many of the functions associated with controlling telephone style devices, including fax, answering devices, and the like. Previous versions of Windows didn't have a standard API set to support operations such as dialing and automatic answering, so application developers had to invent their own.

TAPI addresses this problem with the consequent benefits of standardization and the ability to share devices between active applications.

Underlying many of the features that fall into the communications category is the basic device support offered in Windows 95. Whether you're the owner of a venerable 1200-bps modem or the latest cellular fax device, the communications driver—usually referred to simply as VCOMM—is a critical software component of any connection via these devices. The communications (serial port) driver in Windows 3.1 has been much maligned—especially from the point of view of its inability to handle higher-speed connections. As a result, the developers of many communications applications such as fax packages or terminal programs have replaced the Windows driver with their own. This scattered development has often led to conflicts and bugs that a user of two of the applications has been left unable to resolve. For Windows 95, Microsoft has concentrated a great deal of effort on providing a communications driver that will reliably handle extremely high line speeds.[1] The communications subsystem also benefits substantially from the improvements in the Windows 95 operating system kernel—from preemptive scheduling and dynamic VxD loading in particular.

The design of the VCOMM module follows what has become a popular design technique for Windows components—VCOMM itself is shared among individual ports with hardware dependent operations managed by individual communications port drivers. Each of the standard serial and parallel ports of an ISA machine, for example, would have its own port driver and share the functions provided in the single VCOMM module.

Figure 10-1 on the next page illustrates the main software components that would be present in a Windows 95 system configured for remote communications. Some of the components in the illustration are optional or redundant, and others go by yet more acronymic names. Here's a summary of their functions.

RNA. Remote Network Access is the subsystem that allows a user to dial out from his or her local system and log on to a remote network. The connection is set up so that the network appears to the user just as if he or she had logged on from a directly connected network workstation. RNA includes both a client and a server component.

1. The stated goal is to be able to handle serial line speeds in excess of 38.4 Kbps.

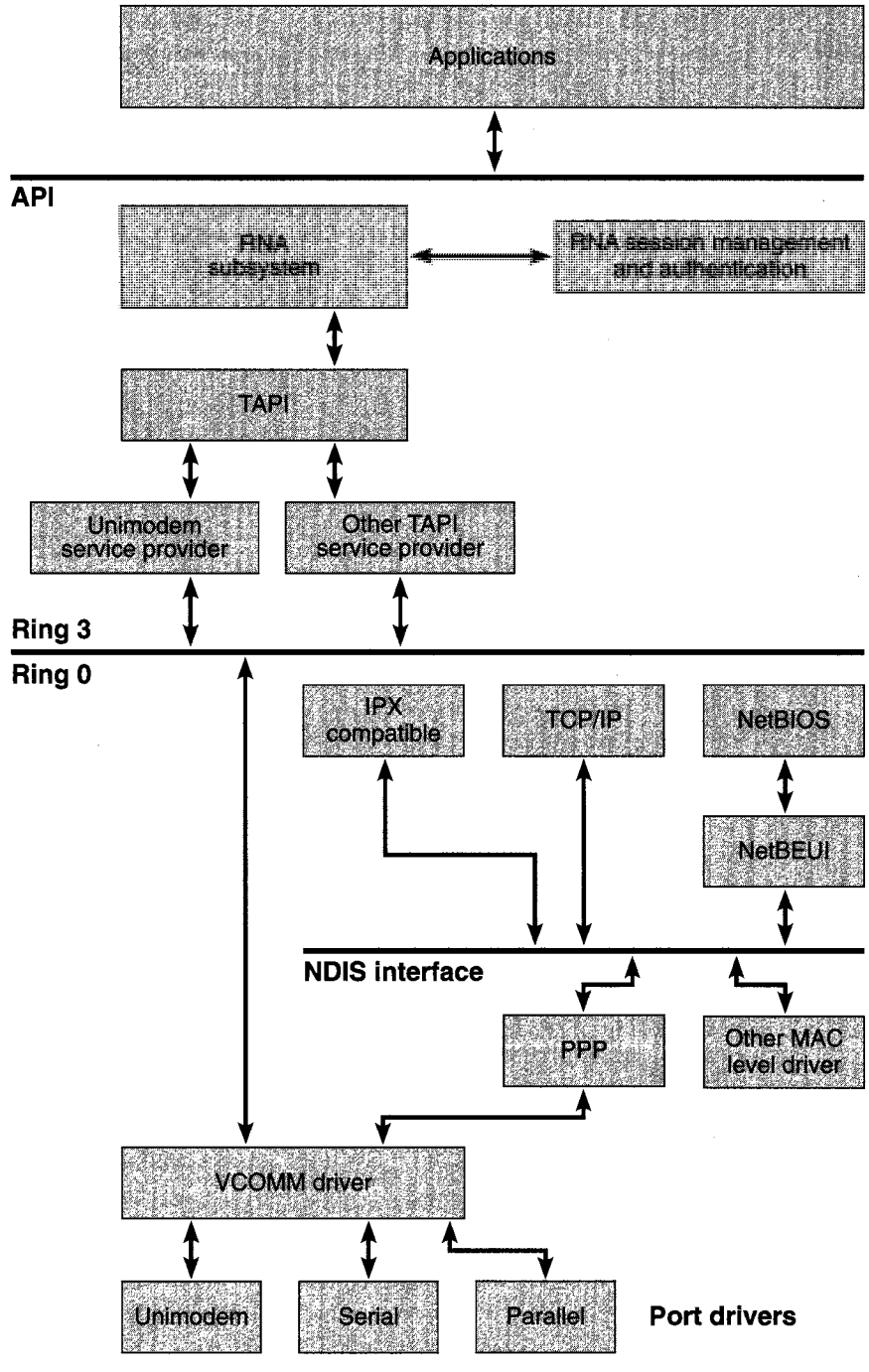


Figure 10-1.
Communications architecture in a Windows 95 system configured for remote communications.

TAPI. The DLL that implements the Telephony API incorporates the new Win32 functions for telephone line management.

Unimodem. The Unimodem service provider is Microsoft's attempt to simplify and unify support for modem devices under Windows. Rather than have each and every communications application developer produce and test its own modem interface, Microsoft has Unimodem use a collection of modem description files to enable every related application to determine a modem's configuration and the appropriate modem control sequences where necessary. In many cases, the application simply uses open and close type API calls and the Unimodem port driver accesses the modem information file.

PPP. The *point to point protocol* driver is for a simple protocol that has been widely adopted. PPP is used for single-session communications over relatively low speed lines (typically telephone lines). The PPP module handles the blocking and deblocking of data packets and simple error correction.

VCOMM. The new communications driver for Windows 95 includes a set of functions intended to be used by the port drivers and other VxD-level clients. The closest equivalent to VCOMM in Windows 3.1 is the serial port driver, but VCOMM addresses additional communication link device types, including infrared and wireless radio connections.

Port Drivers. The port driver components contain the hardware-specific code peculiar to an individual device, such as the serial port, or an infrared connection. Windows 95 will come with standard port drivers for serial and parallel devices. Other port drivers will be supplied by the device manufacturers.

Remote Network Access

RNA refers to the ability of a Windows 95 system to gain access to a remotely located network. The typical scenario features a business traveler equipped with a portable system dialing out from a hotel room to collect electronic mail and other documents from the home office. Many products currently on the market offer this capability. They come in three flavors:

- Dial-in terminal access programs that offer simple point to point connections. On the server side, the software might offer

access to a bulletin board system with file transfer capability or to electronic mail. Commercial networks such as CompuServe and MCI Mail offer this type of service.

- True network access for which the software on the server acts as a gateway to the local network. The remote user can access network resources as if he or she were locally connected. Remote access to network resources is subject to the same security constraints as for a local connection. Microsoft Windows NT offers this feature as part of its Remote Access Services (RAS).
- Remote control software that allows the user to "take over" the remote machine to which he or she connects. The remote user can make use of the capabilities of the machine he or she connects to and transfer files back and forth between the two machines. Products such as Carbon Copy and PC Anywhere implement this capability.

Windows 95 RNA implements the first two of these flavors. An upgraded Terminal application uses the lower levels of the communications subsystem to provide dial-up access.[2] The full RNA subsystem provides network access for remote users using either a Windows NT or a Windows 95 system that has a local network connection. Figure 10-2 illustrates the various network access configurations RNA makes possible.

On the server side, the Windows 95 RNA subsystem supports a single connection, so the most obvious use of this feature will be for a user at a remote location to dial in to his or her own system back at the office or perhaps call back home from the office. In this case, a network might not be involved and the RNA server might simply provide access to the resources of the machine it's running on.

Types of Remote Access

Windows 95 provides three different ways to go about establishing a connection to a remote network:

- Making an explicit connection, in which the user selects a remote system and establishes a session.

2. The new version of Terminal was developed for Windows 95 under contract to Microsoft by Hilgraeve, the developers of the popular DynaComm product.

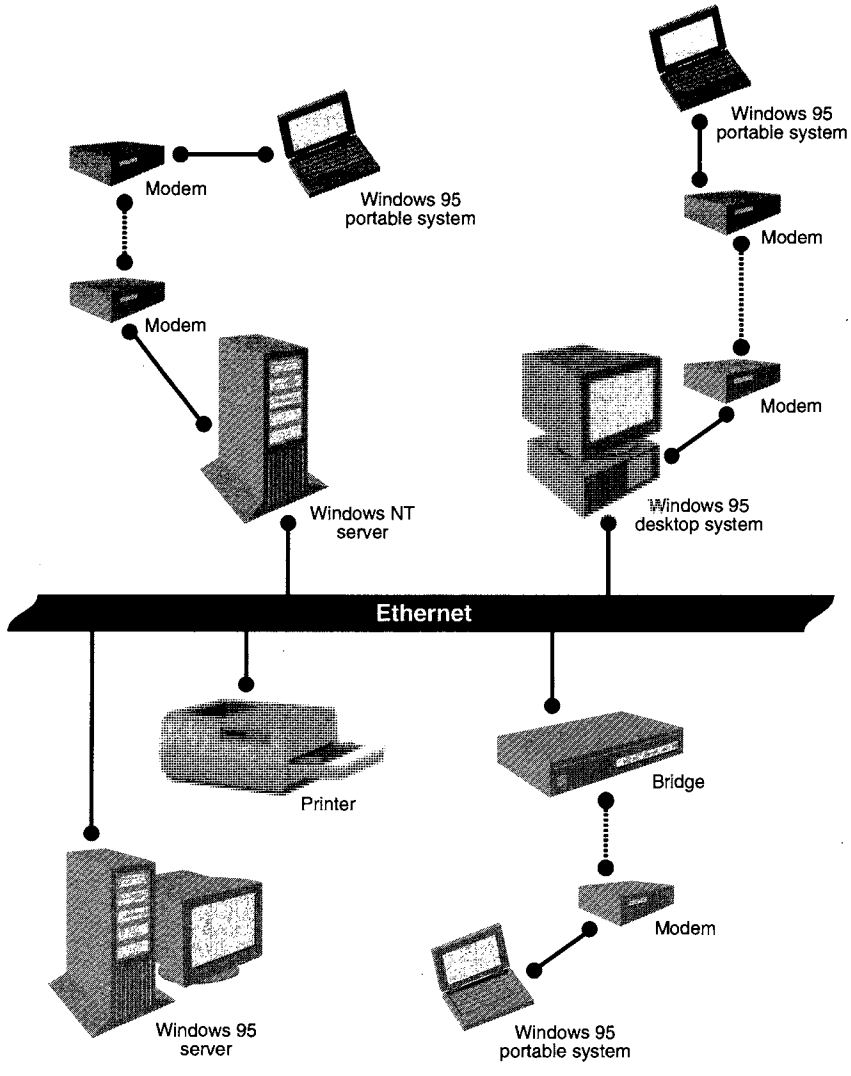


Figure 10-2.
Remote Network Access configurations.

- Making an implicit connection, in which the user tries to access a file or a printer located on a remote system. The Windows 95 shell takes care of establishing the connection with the remote system. Obviously, the local system has to be configured correctly, and the likely delay in getting the connection set up will leave the user in no doubt about what's going on.

- Using the RNA Session API, a set of Win32 interfaces for applications that will set up and manage remote connections directly.[3]

Figure 10-3 is an example of the shell's screen in the case in which the user has elected to make an explicit connection by double-clicking on the Home System icon in the network Remote Access folder. This particular remote system has already been set up with the appropriate telephone number and device to connect through. Once the user has clicked on OK in the login dialog box, RNA takes care of dialing and completing the connection. At the receiver's end, the called system must be running the Remote Access Server (or an equivalent) and be listening for the incoming call.

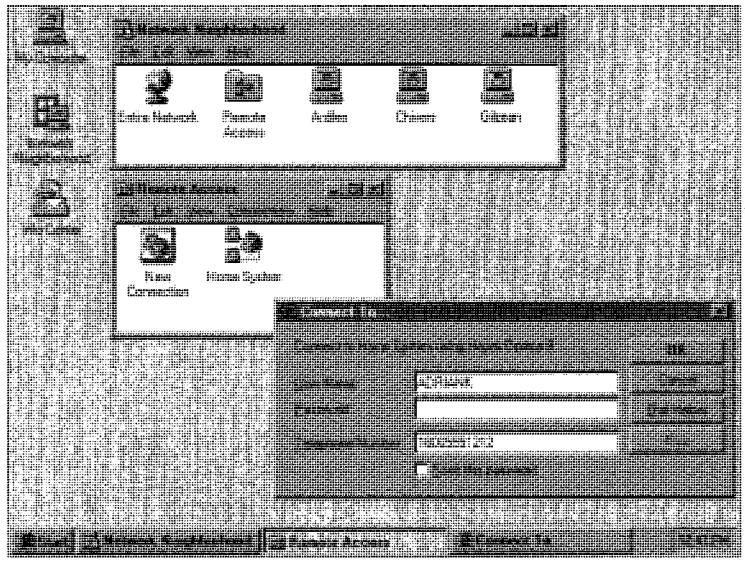


Figure 10-3.
Connecting to a remote network.

Implicit network connections are generally handled by the shell. When the user tries to access a remote resource, the shell initiates the connection attempt with minimal further user input.

The Win32 API associated with RNA provides several functions that allow an application to set up and manage a remote connection:

3. All of the functions in the RNA Session API are identifiable by the *Ras* prefix in their names. There are no equivalent Win16 APIs in Windows 95.

RasDial()	Handles the process of making a remote connection.
RasHangup()	Terminates an active connection.
RasEnumConnections()	Returns information about the currently active connections.
RasGetConnectStatus()	Returns information about the current status of the connection initiated by a call to *RasDial()*.

The Telephony API

The development of the Windows Telephony API (TAPI) began as part of Microsoft's At Work office automation initiative. The intent of the At Work initiative is to integrate common office equipment, such as facsimile machines and photocopiers, with the desktop PC. A PC user could send, receive, and print documents in a common digital format under the umbrella of devices supported by the At Work operating system. The most common device in the office is the telephone, and the At Work effort included the specification of an API that allows Windows application developers to control suitable telephone handsets and conforming exchange equipment. The emphasis for Windows 95 is on what Microsoft refers to as personal telephony applications—essentially applications that assume the use of a single PC and a single telephone handset.

Today most telephone equipment that can be connected to a PC offers the application developer a bewildering variety of (often proprietary) interfaces, and most of the available application solutions tend to be either highly specialized or specific to a narrow range of devices. TAPI is Microsoft's attempt to standardize an interface and, in addition to meeting the challenge of developing a suitable API, Microsoft must convince the telephone equipment manufacturers to support the associated service provider interface (SPI) in the WOSA framework.[4] The use of WOSA allows TAPI to remain independent of the specifics of any hardware device. In the Windows 95 product, the philosophy of multiple providers is retained: for example, a service provider can offer access to a shared network device concurrently with a locally attached device.

For the application developer, the success of TAPI would mean that a single Windows application could be developed to control a wide range of telephone hardware. For the user, the incorporation of TAPI

4. A full discussion of WOSA and the service provider interface (SPI) appears in Chapter Nine.

into the core Windows 95 product ought to mean that there will be a wide range of telephony-related applications available—either specialized applications (call screening, for instance) or applications that are extensions of the functionality available in mainstream desktop applications (the integration of voice mail messaging within an electronic mail package, for instance). RNA itself uses TAPI when it initiates and controls remote connections made over telephone lines.

Telephony Applications

TAPI identifies two separate connection types: a *phone-centric* connection type, in which the telephone handset is directly connected to the telephone network and then to the PC via a serial interface, and a *PC-centric* connection type, in which an adapter card in the host PC connects to both the telephone network and the telephone handset. In the phone-centric case, the application controls the telephone network by sending commands to the handset for forwarding. In the PC-centric case, the combination of the hardware in the PC and the TAPI application software emulates a phone handset to the network and involves the real handset only when necessary.

In the development of telephony applications, these hardware arrangements manifest themselves as a *line* device class and a *phone* device class. A line device is the connection from the desktop to the telephone network. The line device responds to data objects such as an address (the telephone number) and to state changes such as active and inactive. The phone device is the handset component and provides logical access to components such as the ringer and any buttons or indicators on the handset.

One of the important concepts underlying Microsoft's view of telephony applications is the idea that a single desktop machine might run several concurrent applications that have an interest in the single telephone line. An incoming call might be a facsimile transmission, for example, a voice call, or a connection request from a remote modem. An application that conforms to the TAPI interface has to be prepared to examine an incoming call and, if the call is of no interest to it, hand the call off to the next potentially interested application. Similarly, once the telephone line is active, an application that tries to use the line has to be prepared to gracefully handle the error condition resulting from the line's busy status.

Modem Support

First there was a universal printer driver, and now with Windows 95 come a universal display driver and a universal modem driver. Once again, the intent is to provide a common set of well-tested functions that can control a broad range of similar devices. The Unimodem name is given both to a TAPI service provider and to a low-level driver (implemented as a VxD) that works together with a port driver to directly control an attached modem.

There have been other attempts to standardize a modem control interface—notably on UNIX systems. To some degree, the problem is a more tractable one than it used to be since virtually every modem manufacturer uses the Hayes-defined command strings for direct modem control. In fact, the Unimodem driver assumes the standard Hayes command set as a base and then defines exceptions to the command set for specific modems. The description of a modem appears in a text file that might be supplied by the hardware vendor. Windows 95 comes with a large database of known modems—their descriptions are in the MODEMS.INF file, which is a standard component of Windows 95.

When you set up a modem using the Control Panel, the appropriate command strings are copied from either MODEMS.INF or the manufacturer-supplied .INF file into the registry.[5] Once the command strings are installed, the universal modem driver (UNIMODEM.386) can directly access the command strings. An application never sees the command strings used at the lower levels. It merely issues requests such as open and close. This arrangement hides the peculiarities of any particular modem from the application. Figure 10-4 on the next page illustrates the interactions between the various components when a modem attached to a serial port is in use.

Notice that the upper level of the universal modem driver is a TAPI service provider and that it can co-exist with other service providers. At the lower level, the communications driver (VCOMM) routes modem-related calls to the modem driver, which, alone, deals with the registry. For actual control of the attached modem, the modem driver calls back into VCOMM, which in turn calls the associated port driver (the serial port driver SERIAL.386 in this example).

5. You may see references to modem mini-drivers. These are simply the text files that encapsulate the modem commands.

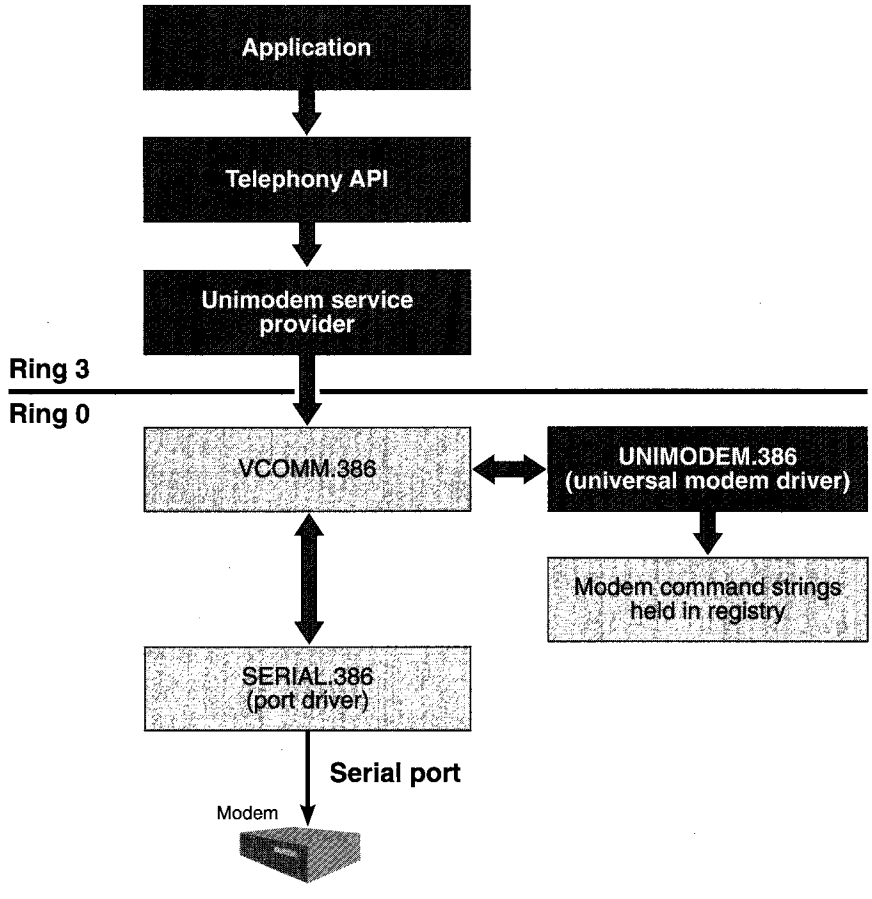


Figure 10-4.
Modem interface.

The Communications Driver

In Windows 3.1, the communications port driver suffered from performance problems engendered by mode-switching back and forth between protected and real modes and by the absence of preemptive multitasking capabilities in the operating system. The VCOMM driver in Windows 95 helps to solve the performance problem by providing a protected mode code path from the application all the way to the hardware. And the improvements in the OS itself assist in meeting the goal of reliable, high-speed communications device support.

Figure 10-5 illustrates the way in which VCOMM interacts with other system components. Notice that the COMM.DRV module is

there to provide compatibility for existing Win16 applications. It is simply a thunk layer that translates 16-bit API calls to the Win32 interface. It is not an updated version of the Windows 3.1 communications driver.

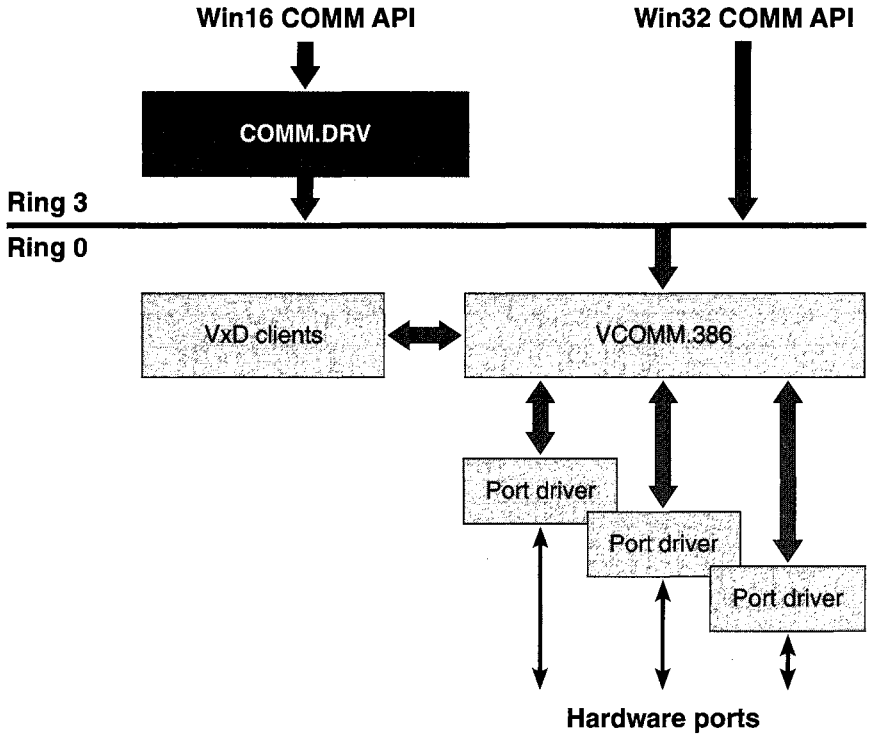


Figure 10-5.
Communications driver components.

VCOMM is a static VxD that is always loaded during the Windows 95 boot process. VCOMM participates in the Plug and Play subsystem by loading the appropriate enumerator and subsequently loading the individual port drivers (which are dynamically loaded VxDs) as ports are first opened. VCOMM is multithreaded, and its code is shared among all of the lower-level port drivers that interact directly with the hardware. The VCOMM services are available to other VxDs, but they are never called directly by an application, only via the defined Win32 APIs.[6]

6. All of the VCOMM services can be identified by means of the prefix _VCOMM_.

Windows 95 provides port drivers for both serial (SERIAL.386) and parallel (LPT.386) ports. When VCOMM first loads a port driver, the driver registers its presence using the _VCOMM_Register_Port_Driver service and provides the address of a *DriverControl()* function in the port driver. VCOMM uses the *DriverControl()* entry point to instruct the driver to carry out the function of initializing a hardware port. Once a port is recognized and registered, VCOMM will open it using a *PortOpen()* function in the driver. Subsequent calls from VCOMM into the driver go via a table of functions whose address is returned to VCOMM as a result of a successful *PortOpen()* call.

The Info Center

Quite late in the development of Windows 95 Microsoft decided to group the various information access components under the collective name of Info Center. Although the name has little more significance than to be a simple way to refer to the collection of information access modules, it's an umbrella for a useful grouping. The structure of the Info Center suggests that its capabilities can be broadened significantly in the future. Thus, the early establishment of a "brand name" for these Windows 95 functions seems to have been a good idea. Competitive issues are at work here too. One of the major challenges to Microsoft's dominance of the office software market has been the Lotus Notes product. Positioning the Windows 95 Info Center as a key component of a workgroup application strategy allows Microsoft to begin reclaiming some of the ground it has lost to Notes. Synonymous with the Info Center is what Microsoft calls "messaging," and you'll hear talk of "messaging APIs" and "messaging services." The messaging APIs and services are at the heart of the Info Center.

The Windows 95 Info Center serves as a common access point for the applications and services that deal with office information—electronic mail messages, voice mail messages, facsimile documents, standard forms, and other types of typically textual, loosely structured data. For the user, Windows 95 provides a Microsoft Mail client and the Internet access tools that rely on the WinSock API and the TCP/IP protocol stack.[7] For the application developer, the underlying services provide a standard interface to various messaging systems. The structure of the

7. The latest versions of Windows 95 actually have an Info Center icon on the default shell desktop—similar to the local computer and network neighborhood icons. The similarity suggests that the Info Center will be a commonly used information access tool.

Info Center allows applications and service provider modules to be added very easily. Figure 10-6 illustrates the components that Windows 95 groups under the Info Center heading.

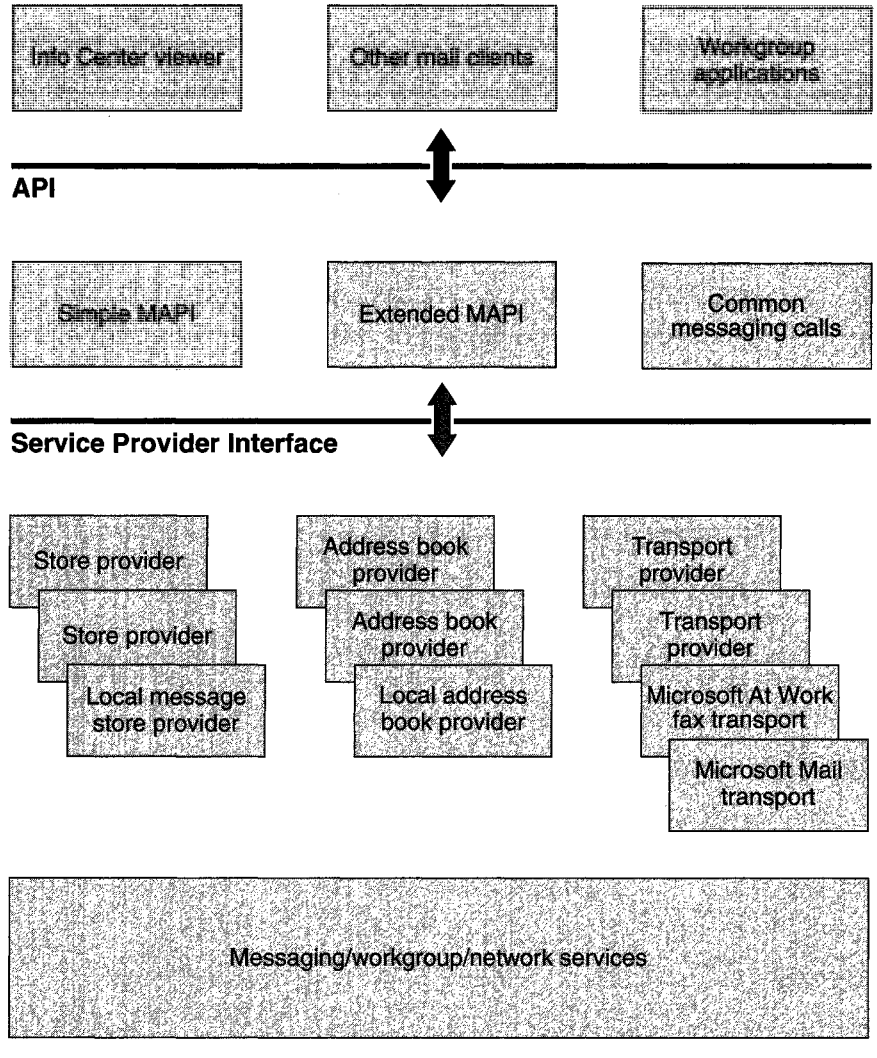


Figure 10-6.
Info Center architecture.

The Info Center breaks into three layers of software: the application level visible to end users, which includes an electronic mail application, for example, and two lower layers. The first of the two lower layers

is a collection of Windows DLLs that implement the messaging APIs, and the other of the two lower layers is a service provider layer offering access to different message-related services. Once again, the structure conforms to Microsoft's WOSA model. Below the service provider layer can be any network protocol and transport or, in the case of voice mail handling, for example, some other subsystem such as TAPI.

Info Center Applications

The Info Center viewer is a Microsoft Mail client integrated with the Windows 95 shell. Any time you're using Windows 95 you can send a message—there's no need to start up a separate e-mail application. Microsoft has also announced that Windows 95 will include an interface to the Internet, although by mid-1994 the final form of this application hadn't been determined.[8]

If you're in an organization that has standardized on a non-Microsoft electronic mail package such as ccMail, the inclusion of the Microsoft client application won't really help you. But, as you'd expect, the messaging API is available to all applications, so Windows 95 will no doubt have a variety of electronic mail packages available for it.

Although other kinds of applications don't strictly come under the Info Center umbrella, the inclusion of the messaging API as a standard component of Windows 95 means that other applications—word processors, for example—can make use of the messaging services. An application that deals with documents can add a Send Document option to its standard menu and enable direct document transmission using the messaging APIs. Microsoft refers to this type of application as "messaging aware." This isn't new. Many applications have offered this feature under Windows 3.1. The difference is that the messaging APIs are now a standard part of Windows 95, and any application can rely on their presence.

Messaging APIs

The messaging APIs in Windows 95 are incorporated into three separate modules, two of which implement Microsoft's core messaging effort—the Messaging Application Programming Interface (MAPI). Although Microsoft has gathered support from other companies for MAPI, the

8. Including this feature was a late decision, spurred by the growing public interest in the so called information highway.

design and development of MAPI are very much under the control of Microsoft. These are the three components of the messaging API:

Simple MAPI. The basic send and receive functions of MAPI.

Extended MAPI. A superset of Simple MAPI that incorporates message storage, retrieval, and searching capabilities.

CMC. The Common Messaging Calls, a Windows 95 implementation of the functions defined by the X.400 API Association, of which Microsoft is an active member.

Both MAPI and CMC allow an application to use a standard set of functions for messaging. The application developer doesn't have to worry about the details of the underlying message system. The essential difference between MAPI and CMC is that MAPI is defined for Windows systems only—Microsoft hasn't made any attempt to adapt it to other operating systems. CMC on the other hand is defined as OS independent, and if you're planning a messaging application for a variety of different hardware and software environments, CMC is preferable to MAPI. In terms of their basic functions, CMC and Simple MAPI are very similar.

Simple MAPI contains only 12 messaging functions, and it's intended primarily for use in messaging aware applications rather than for the implementation of a full blown messaging application—an electronic mail package, for example. The Simple MAPI functions allow an application to send and receive messages and to manipulate message address information. Simple MAPI also allows files to be attached to messages and OLE objects to be incorporated in messages (hence the Windows dependency).

Extended MAPI is intended for major messaging applications—electronic mail systems, workflow applications, and forms management packages, for example. Functions in Extended MAPI allow the application to access and manipulate the message store and the address books supported by the service providers and to incorporate forms management capabilities.

Messaging Service Providers

Underlying the messaging API is the set of service providers that understand the details of the messaging system they manage. All of the providers support the same service provider interface, but each service

provider is written to interface to a particular messaging system. So, for example, one service provider will support Microsoft Mail on the local network whereas another could support dial-in access to MCI Mail.

Common to the design of each MAPI service provider are the notions of a *store provider* (wherein information can be stored and retrieved), an *address book provider* (offering some means of translating a name into an address), and a *transport provider* (which takes information and actually transmits it to the intended recipient via some physical means, such as facsimile transmission or simple file copying). This separation of duties is masked by the messaging API, and, in fact, the underlying service provider can be implemented as a single module.

Microsoft plans to include a personal address book provider and transports for the At Work FAX interface and for Microsoft Mail. The local address book in a Windows 95 system is the single place where user names and associated information are collected. The networking system, for example, uses MAPI as the means for acquiring user information and translating login names.

Portable System Support

Microsoft's standard gee whiz demonstration of Windows 95 portable computer support comes in a segment in which Plug and Play gets the spotlight. The scenario involves an imaginary user removing his laptop system from its desktop docking station and rushing off to another location. This user doesn't bother to turn the laptop machine off, and while he heads out to the waiting taxi, the Plug and Play subsystem dynamically reconfigures Windows 95 so that the user can return to his word processing session as soon as he takes his seat. Do you know anyone who might do this? Neither do I. Nevertheless, as a technology demonstration, it's gripping stuff. Cynicism aside, Windows 95 does include a number of features specifically intended to improve the use of portable systems. Most of these features rely on aspects of the Plug and Play subsystem, and generally the user doesn't have to worry about what's going on—it just works.

Power Management

One of the well-researched technologies in the last few years has been the power supply for portable systems. Low-power chips and displays and im-

provements in battery technology have combined to make battery-powered machines feasible for even long trips by air. These hardware improvements must work hand in hand with software enhancements that allow the user to control the system, and many portables come equipped with a utility for customizing power consumption. Nowadays, it's the user who controls the length of the interval before the screen blanks or the hard disk spins down to an idle state. In the Plug and Play subsystem, these functions are subsumed under its power management activities.[9]

Docking Station Support

Although portable systems with docking stations haven't sold in the numbers that were first predicted, Windows 95 may be the catalyst to change that. The situation that the Windows 95 Plug and Play subsystem needed to handle is exactly the one described in the earlier example—how do you go about dynamically reconfiguring a system when it moves between a docked state (presumably with access to a network and with a good, high-resolution display) and an undocked state (with a portable display and perhaps a different pointing device)?[10]

Plug and Play is key to solving this problem. The automatic reconfiguration of the system involves unloading and loading the VxDs that control the attached hardware. As a device disappears, Plug and Play will unload the controlling device driver. If a device changes (an external 1024 by 768 256-color display becomes a local 640 by 480 16-level gray-scale LCD screen, for example), the system alters its configuration to suit. The reconfiguration isn't just a system-level activity. Plug and Play will broadcast messages informing running applications that the configuration is about to change. The applications can respond by closing files, blocking the system reconfiguration process, or simply terminating. If the system's FAX card is about to disappear, for example, the background FAX receiver application has no reason to continue to run. For more subtle changes, such as the change of display described above, the application will have to recognize the difference in capability and react accordingly.

9. Details of the state of the art in power management are to be found in the *Advanced Power Management Specification Version 1.1,* available from Microsoft.

10. Microsoft also intended to implement deferred printing in Windows 95—so that even if your printer is not currently attached to your machine you can go ahead and print. The physical output will appear when your machine is next connected to the printer. As of the Beta-1 release, this feature hadn't been implemented.

The reconfiguration process is most likely to take place at power on. You'll turn your machine off, pull it out of its docking station, head out of the office, and power the machine on sometime later. The machine will boot up in its new configuration. This won't be the case with PCMCIA peripheral cards: one likely operation is to remove one card and replace it with another of a different type while the system is running. Windows 95 will manage this reconfiguration process the same way it does at power on, and, after a short delay, the system will be reconfigured with no user interaction. You finally have a good reason to fill your pockets with PCMCIA credit cards whenever you head out of the office.

File Synchronization

One irritating problem that comes up when you're using two different systems is needing to ensure that you're always using the most up-to-date version of a file. If you have a single portable computer and docking stations wherever you go, you've solved the problem. But if, like most people, you copy files from one machine onto a diskette and then copy that diskette's contents onto another system, you're always running into the problem of synchronizing the two different physical copies of the file. Windows 95 has a "briefcase" that makes it easy to manage updated copies of files.

The shell allows you to create a briefcase object and drop other objects into it. When you leave the office, you simply copy the entire briefcase to a diskette (or perhaps across the network to another hard disk). You can work on the files in the briefcase and then get the shell's assistance when it's time to return any updated copies to the original system. Typically you'll create a briefcase on the desktop and leave it there, although you can create many independent briefcases if you want to. In the example shown in Figure 10-7, the file CHAP10.ZIP has been copied from the desktop to the briefcase. The original remains in place.

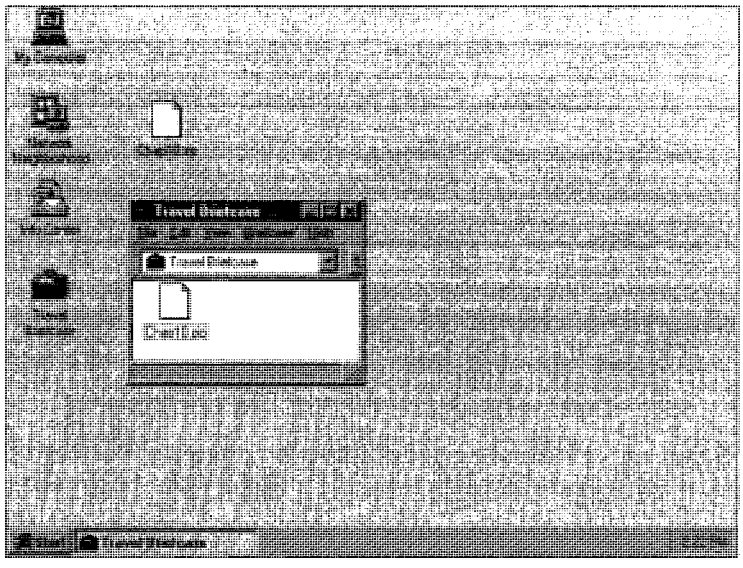


Figure 10-7.
A briefcase on the desktop.

You copy the briefcase and its contents by simply dragging and dropping the whole thing to its destination. In this example, the destination is a floppy disk. Examining the contents of the briefcase on the disk would lead you to believe that only the files you copied to the briefcase are present in the briefcase (see Figure 10-8). In fact, the shell adds hidden files that describe the contents of the briefcase to assist the later reconciliation of different versions of the files you've copied.

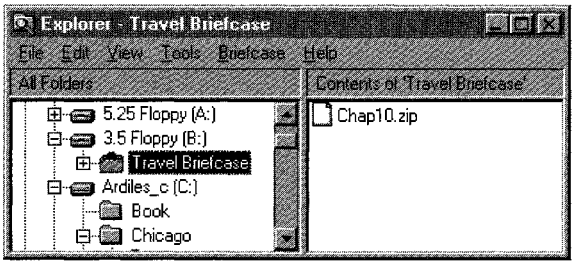


Figure 10-8.
The contents of a briefcase.

When you return to the original system, you copy the briefcase back to the desktop and then initiate an update operation on the contents of the briefcase. The shell compares the versions of the files and recommends the reconciliation action that seems to be appropriate. In the example shown in Figure 10-9, the shell suggests that the updated copy of the file contained in the briefcase ought to replace the original file on the desktop.

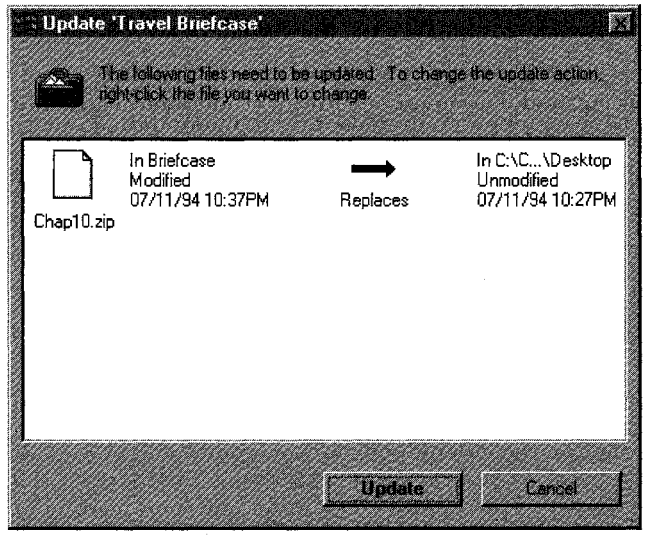


Figure 10-9.
Replacing a file with an updated version from the briefcase.

Of course, if you are only one of a number of people working on a shared document, it's possible that the original will also have been updated in the meantime. In this situation, the shell won't know how to proceed, and you'll see a dialog similar to the one shown in Figure 10-10. At this point the user has to guide the update process.

Although this is a simple scheme, in practice it works well, and naturally there is more to it than simple file modification date and time comparison.

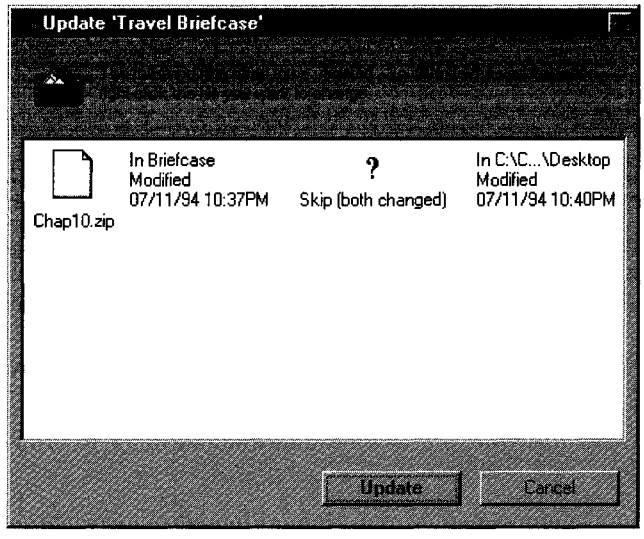


Figure 10-10.
Reconciling a file when both the briefcase version and the original have been modified.

The Briefcase API

Both briefcases and their contents are controlled by the *AddObjectToBriefcase()* API. This API not only copies the physical data associated with the document to the briefcase but also updates the control information associated with the briefcase. Objects copied to the briefcase in some other way won't have this control information incorporated and thus can't be reconciled at a later time.

The *ReconcileObject()* API initiates the process of reconciling two different copies of an object. The shell calls on the services of a *reconciliation handler* to perform the actual updating process. In many cases this will simply mean copying the newest version of the file over the older one. But in cases in which a true merge of the file contents has to take place, the reconciliation handler must understand the details of the file format it's dealing with. Microsoft plans to provide a number of standard reconciliation handlers for common file types.[11] An application can also

11. Although this was announced, the exact plans were still vague as of July 1994. Also, an earlier announcement that objects within OLE compound files could not be individually reconciled appears not to be true, so expect this capability as part of the Windows 95 product.

register its own reconciliation handler and thus be called on by the shell to perform the reconciliation action for the associated object type.

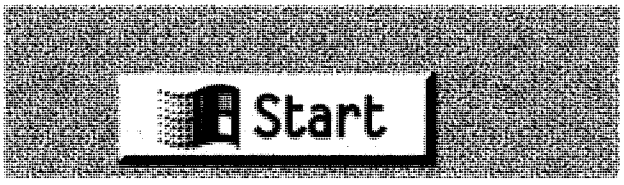
Conclusion

If Windows 95 meets Microsoft's dual goals of providing excellent communications capabilities and providing good performance on existing 386 machines with only (sic) 4 megabytes of memory, it will be a strong contender for adoption as the preferred OS for portable and home computer use.[12] If the lower-level communication drivers live up to the advance performance claims, there should be no barrier to developers basing their communications software on Windows 95. With the layered network architecture and MAPI, Windows 95 should provide a great platform for remote networking and applications that rely on electronic mail and other connectivity options. Windows 95 also addresses a few of the real practical problems of mobile computing: the synchronization of files, deferred printing, and (with Plug and Play) the dynamic adjustment of system configuration.

At one time I planned to discuss the capabilities of Windows 95 with respect to handwriting recognition and the use of handwriting recognition technology on the so called personal digital assistants (PDAs). The early Chicago presentations gave significant airtime to the handwriting technology planned for Windows 95, but the industry's love affair with pen-based systems has cooled off in recent months. Microsoft still plans to incorporate handwritten input recognition as a standard part of Windows 95, and the WinPad application is intended principally for use with a PDA. It doesn't seem likely that Windows 95 will usher PDAs into a new era of productive use—but we'll have the basis for some exciting applications when and if handwritten input becomes practical.

12. Naturally the other part of the home equation is what Windows 95 will offer game players and developers. Microsoft's announcement of the WinG graphics library and its recent efforts to court MS-DOS game software developers ought to help meet this particular need.

Although I've examined much of Windows 95 in a lot of detail, I've passed over some features, and other features are still changing as this book goes to the printer. In a concluding interview, I had a chance to ask Microsoft's Paul Maritz, Senior Vice President, Systems Software Division, and Brad Silverberg, Vice President, Personal Systems Group, about late-breaking news and Microsoft's goals and aspirations for the product during the latter part of 1994 and into 1995.



EPILOGUE

LEAVING CHICAGO

By the time this book went to press, the Beta-1 release of Windows 95 (née Chicago) had been distributed to about 15,000 developers and users around the world. Early reviews and product evaluations had appeared in industry magazines, and interest in the product had already swelled beyond the dull roar level. The early sightings of the product also raised a number of questions—about the positioning of Windows 95 vis-à-vis Windows NT, about the new user interface, and about the likely level of success for Windows 95.

Right before this book went to press, I talked with Paul Maritz, senior vice president of Microsoft's Systems Software Division, and Brad Silverberg, vice president of Microsoft's Personal Operating Systems Group—the group directly responsible for Windows 95. The interview took place in Paul's office at Microsoft on July 22, 1994. I asked Paul and Brad about their aspirations for Windows 95 and about some of the product features already receiving critical review. Their answers were candid and largely devoid of the marketing hype that Microsoft is so justly famous for. Brad in particular is an irrepressible Windows 95 enthusiast. Clearly, neither man had any illusions about the amount of work still left to do before Microsoft would be in a position to ship a great product, but their demeanor suggested that the light they saw at the end of the tunnel was not from an oncoming train. Here is the interview. It's been edited for syntax, and the sounds of lunch have been deleted, but the semantics remain untouched.

AK: Adrian King, Interviewer
PM: Paul Maritz, Senior Vice President, Microsoft Systems Software Division
BS: Brad Silverberg, Vice President, Personal Operating Systems, Microsoft

AK: *My first question relates to the potential for confusion when Chicago appears in the market. You'll have a Windows 3.1 product that's been very popular, Chicago, Windows NT, and Cairo coming up. As far as the evolution of the desktop is concerned, over what time frame do you see which operating system claiming the major share of the desktop market? And what should people be doing when they upgrade or when they really need to move to the more powerful product?*

PM: There are basically two ways you can approach that question. One is Chicago vs. Windows 3.1, and the other is Chicago vs. Windows NT. I'll let Brad address the 3.1 part of it.

BS: Chicago is simply the next major version of our high-volume desktop Windows operating system. So it's the successor to, replacement for, Windows 3.1 and Windows for Workgroups. Those products have been phenomenally successful. We're selling over 2 million units of those a month. We announced yesterday that we've shipped over 60 million copies of them. And Windows Chicago is just the next version. Anybody who will be buying a new version of Windows after Chicago comes out should be buying Chicago. Anybody who is running Windows should be running Chicago. Just as today I don't know anybody who is running Windows 3.0, I would expect in some period soon, maybe a year after Chicago ships, that if you talk to people who are running Windows—they'll be running Chicago. It's a replacement. And it's complementary with our version of Windows targeted for high-end workstations, mission-critical applications, technical workstations, and the most demanding corporate applications. That's Windows NT. Daytona is simply the next version of Windows NT, and Cairo is the next major version of that product line.

PM: I think there will come a day when we will shift more and more of our corporate customers toward the NT platform. But with Windows NT we deliberately bit off some very challenging things. Basically it's a tremendous investment in raw software technology—writing a code base that's truly portable across architectures, that's certifiably secure, that's suitable for distributed computing, that's highly extensible, etc. And all of those things come at a price. They require a lot of resources, which means that as of today Cairo is really targeted at the higher end of the line—to people for whom those features of security, extensibility, and scalability are very important, and who are willing to pay for the hardware resources necessary to allow them to have those features.

Over time, as the center of gravity in the hardware base shifts, particularly in the corporate environment, as people move toward Pentium-class machines, with 16 or more megabytes of memory, we'll be able to shift more of our corporate customers in the direction of the Windows NT code base. But we see forever having to maintain at least two implementations of Windows in order to be able to cover the broad spectrum of people who use PCs.

BS: The products represent two natural design centers, and that will continue. I mean the natural flow of technology is always—starts out at the high end, a couple of years later it becomes mainstream, a couple more years later it's obsolete. It's no different from what we see today.

PM: Today and in the future we see ourselves having a design center at the high end, where we're trying to push technology as fast as we can, realizing that we're probably using more resources than most people have in order to do that. On the other hand, we need to remain really focused on the broad market in two senses, making sure that we stay within the resource constraints that not only new machines but the installed base of machines has and that we stay very focused on producing software for ordinary people who don't want to understand anything complicated and just want to use their systems.

We see ourselves having to maintain these two design centers and two teams focused on doing that. That's been our strategy for the last three years, and I can see that as being our strategy in the future. What you're seeing is simply the output of those two focuses coming into the marketplace when we move from Windows 3.1 to Chicago, and there will be successors to Chicago. Some of those successors to Chicago might use a lot of the technology that you find only in Windows NT today, but they'll still be, from a design point of view and a philosophy point of view, targeted at a broad mass market. At the same time, we'll be using new technology at the high end—what you think of as the Windows NT line—where our focus is really on client-server computing, distributed computing, system administration, and a lot of other aspects. We hope we can increasingly share technologies between those two environments, but I think there's always going to be a difference between them.

It is a more complicated strategy, both to explain and to execute. It certainly does put some strains on us, but I think the result of it is that we'll be able to serve a broader class of customers in the future and not

be forced to bifurcate the world and say that for corporate computing you use only Windows, and for home computing you have to go and buy some other random product that comes out of Nintendo-space or whatever.

BS: That's like with Intel when the 386 first came out. It was high end; you only ran the 386 for servers; and then it was on high-end desktops; and now it's pretty ubiquitous. And now, at least from an accounting standpoint at Microsoft, we've written off all our 386s. That's just a natural flow of technology. But there's still that high-end space. Intel is still producing very high end chips, and they are focused on the server first, and then they come down to the desktop. The hardware technology flows that way. You'll see the same thing in our operating systems.

PM: There are some things that flow the other way as well. Ease of use factors in particular. And that's what you see being pioneered in the Chicago area. Things like the new user interface and the Plug and Play framework, which are absolutely vital for the broad market but which you'd like to have in the business-oriented market and the high-end market as well. And those things will flow into our high-end product line and be used there. So Cairo has as one of its objectives to absorb some of the features that are being introduced with Chicago.

AK: *At least in the Windows NT product line, you've made a big investment in the portability of the code for adaptation to RISC processors, which is not a consideration for Chicago. Yet the RISC-based machines have had a minimal impact in the market so far. Do you see that changing? Or do you think Intel—Intel-compatible chips—is going to hold sway forever?*

PM: It's still hard to say. I mean, today, clearly Intel has been very successful in bringing new parts into the marketplace and increasing their price performance on a regular basis, which has meant that it's been tough sledding for anyone else to make enough of an impact to get some market share. But we still think we've done the right thing in terms of slowly but surely investing in technology that says, whenever, whatever happens down in the silicon, our customers are going to be insulated from it; that we can take advantage of innovation wherever it comes from; that it's not something that people need to be concerned

about. I think Intel is very focused on the challenge posed to them by the Power PC chip. I think the huge investments that they're making in future processors, the kinds of deals they're announcing with companies like Hewlett-Packard, mean that they have every intention of not giving up their leadership.

AK: *For each of these products, and here I mean Chicago, Daytona, and Cairo, what's a good configuration for me to buy to run them?*

BS: What applications do you want to run?

AK: *Microsoft Office?*

BS: The goal with Chicago, and one we've worked super hard as a team to achieve, is that whatever you're running today, on Windows 3.1, if all you do is move from 3.1 to Chicago, you'll be at least as happy as you were before. So that the performance you saw before when you ran those applications you'll see with Chicago.

PM: And on higher-end machines you'll be even happier.

BS: The Chicago performance curve is that the more memory you add, the better we can really take advantage of it. And that is something a little different from 3.1. In Windows 3.1, we weren't able to take advantage of higher amounts of memory the same way, and the performance curve would flatten out. But with Chicago we have an integrated cache management system for the filesystem, the network, and virtual memory that allows us to dynamically balance the cache in real time to really take advantage of additional amounts of memory. But if you're running games or Microsoft Works or Microsoft Publisher, as with a lot of these home machines, and you go to a mass merchant like Costco, what you need and what they sell is a 4-MB machine. People take it home and they're happy. How many? Seven million home machines sold in the United States in 1994? People are buying 4-MB 486 systems for their homes.

PM: If you run some of the application benchmark suites that use normal features like cut and paste, printing, and things like that, with Chicago the knee of the performance curve is approximately 6 MB. For that user scenario you won't get a lot of performance increase by going

above 6 MB. And on Daytona *[Windows NT version 3.5—Ed.]* the knee is around 12 MB.

AK: *What is it going to be with Cairo?*

PM: You can't say at this point in time. Clearly the development team is going to work hard to make it as good as it can be. Both our teams, the Chicago team and the Windows NT team, have learned the religion of "you'd better stay on top of size and performance." It's very hard to put those things back into a product. You have to stay on top of them up front. The Cairo team—they're going to be working really hard trying to contain that. On the other hand, their goal is to be a very functional platform, so they have to set the trade-off dial in terms of resources vs. function. And it's set differently on that platform. And the kinds of customers who will buy Cairo are not nearly as concerned about whether it runs on a 4-MB machine.

BS: One of the missions of Chicago is to be able to upgrade the existing installed base. It's not just for new machines. That means. . .

PM: You've got to be religious about it.

BS: . . .you've got to be really hard core about making sure you run on what people have today and not have to have them buy more memory. And that means running well with whatever they're running today, and running in the same amount of memory. At the same time, I'm sure as people get into Chicago, as they want to start taking advantage of some of the new capabilities, sure they'll need more memory. As you take advantage of stuff you weren't using before, you might need additional resources.

PM: I think the other thing to say is that usage patterns of applications are changing as you go toward compound documents and things like that. You really have to have a lot more memory than many people do today. We're rapidly reaching the day when applications' usage of memory is getting to dominate the operating systems' use of memory. To really answer those "What configuration?" questions, you have to ask, "What kind of applications? How many? How complex are your interactions among them?"

AK: *My follow-up question would be, given that there's this big emphasis on OLE. . .*

PM: There's no question that if you want to get the full benefit out of OLE you've got to have more memory. If you really wanted to use one of the modern office suites, whether it be Microsoft Office or Lotus Smart Suite, to its fullest capability, you'd be looking at an 8-MB minimum machine.

BS: For that type of system. Some people are very content to run Works or Publisher or run their games. There are millions and millions of people like that.

PM: Or even within the suites, they may be using something but not using OLE. Perhaps just doing basic word processing, for example, so they don't need all that extra memory.

AK: *So if I walk into Computer City in a year's time to buy a new system. . .*

PM: You personally? Oh, 32 MB easy. . .

AK: *No. I'm buying it for my mother or somebody. Is Computer City going to have 8-MB machines as their standard boxes on the shelves?*

PM: In a year's time? I think so.

BS: Probably that will be typical for Computer City. Costco might still have quite a selection of 4-MB machines. Not as many as today. But Chicago won't be a factor in that.

PM: Brad and I were talking about that this morning. PCs, I mean really well-equipped PCs, 486-class machines, are almost down into the consumer appliance price band. And it's interesting to speculate about what happens when a decently equipped multimedia machine gets below $800. We might see a whole new segment of the market open up there. Which is another reason we have to remain very, very focused on assuring that we'll have software that continues to run on the 4-MB level for some time to come.

AK: *Talking about Chicago in particular, what I've noticed most as I've used it during all the testing periods is the amount of effort that has been applied to cleaning up everything in Windows that used to annoy you. I mean, every little detail has been gone into. There is no stone unturned. That plus the new features represents a huge amount of development and testing effort—in particular, compatibility testing. Given that you're now later than you would have liked to have been, in terms of releasing Chicago, do you regret any of that investment?*

BS: Oh no. No. That's Chicago's mission—first and foremost to make PCs really easy to use, delivering on the promise of PCs as an appliance. That's the number one thing we set out to do with Chicago.

There were really four things we set out to do in Chicago. One was to make PCs easy to use. That involves a new shell, Plug and Play, and this fit and finish polish you've just talked about. Number two is to have a modern 32-bit operating system underneath with threads and 32 bits and all that stuff. An aspect of that is to make Chicago a fully bootable, complete operating system so that it's not limited by DOS, not crippled by DOS, and has all the benefits of being a completely self-contained graphical operating system. The third element was connectivity—whether in a LAN or a WAN or a mobile dial-up environment. And the fourth is compatibility: being a no brainer upgrade.

Clearly number one was ease of use. And that was the thing that drove a lot of the things in category number two—the powerful operating system. For example, we added long filenames. When we set out to do Chicago, we didn't think we could figure out a way to do long filenames, in the FAT filesystem, in a compatible way. For years, I mean you know this, we've continued to look at this problem. The idea of long filenames is not a new one. Eight-dot-three names is not something that people have always said, "Wow, this is a really great thing. Let's stick with it." It's really painful. But every time we've looked at it and had good people look at it, they've failed to come back with solutions that were workable. But this time, when they came back and said, "We can't figure it out," we sent them back and said, "We don't have a product unless you fix that." I can't imagine coming out with the next major version of Windows, whose mission is ease of use, and we're still telling people they need to use eight-dot-three names. That's failure. So we went back, and the team came up with a very, very clever solution that allows us to have eight-dot-three names as well as long filenames in a compatible high-performance way. I think it shows the commitment to solving hard technical problems in the kernel that is one of the de-

fining characteristics of Chicago. So I don't regret those efforts for a second. Chicago is going to last for a long time. The legacy of Chicago is going to be with us for years. And cutting corners to release the product a month or two earlier would have been a completely false economy.

AK: *Are there features you wish you hadn't included? For whatever reason? You don't like them. You don't think they're applicable in the current market. . .*

BS: I love the product. I'm so in love with this product. My history of using the product is that I have two identical machines in my office. Both 8-MB 386, 33-MHz systems. One runs Windows for Workgroups 3.11 and the other has been running Chicago since M5 time frame *[December 1993—Ed.]*. I wanted to be like a user and use the product like a user. So initially I spent most of my time, probably 80 percent of my time, on the WFW machine, and then I would just go over to Chicago and explore for a while and find some things I didn't like and send some mail to see if we could get this or that fixed. And as the product progressed, it got better and better and faster and easier and more robust—to the point now where 99 percent of my time is spent on the Chicago machine. When I have to go back to the Windows for Workgroups machine, it's like, "This is the old stuff. How did I ever use this? How did I ever like it?" And I think the shell team has done a phenomenal job of really delivering on the promise of ease of use—it becomes addictive, so much so that you just don't want to use the old stuff anymore. And Windows 3.1 really is, in comparison with Chicago, last generation. So, I can't really point out anything I wish we would have done differently. I wish, obviously we all wish, that the product was on the market today and we were working on version 2. But we're committed to making sure the product is right before we ship it.

AK: *Let's talk about the user interface some more. Already, in some of the reviews of the first beta release, there's been criticism that the shell is too different or simply a mix of lots of other things that have gone before. What's your response to that, and what do you think are the really original features of the shell?*

BS: I think the shell is tremendous. And the feedback I get from beta testers, the vast majority of beta testers—and I'm very active on the CompuServe beta test forum, I know these people, I've worked with them for years, and they don't hold back—what do they think? They

love it. You know, the first day it feels like a new pair of shoes. It feels a little bit uncomfortable. You're just not used to how it feels. The second day it starts to get a little broken in. By the third day it feels like the most comfortable pair of shoes you've ever owned and how did you ever wear the old ones? Some of the people who are passing opinions haven't even used it! There are other people, who for whatever reasons, want to stick with the old user interface, for training or migration reasons, maybe. That's fine. We're glad. We'll supply that feature and we'll make it easy for people to use File Manager, Program Manager, and so on. And they can migrate to the new user interface at the pace they like.

I have heard some of the criticisms, that it's a collection of OS/2 and Motif features, and features from all these other things, and it just makes me laugh. We never even looked at Motif. I can't tell you what Motif looks like! I don't think Joe can either *[Joe Belfiore, the lead shell designer in the Chicago group—Ed.]*.

PM: There were people who looked at Motif. We didn't put our heads in the sand and not look at what was going on around us. But what is certainly the case is that this thing was not designed from "Oh yes, let's take three features from there and three features from there." It was designed to solve problems that had been identified in the existing Windows 3.1 user interface.

BS: And problems in other graphical operating systems.

PM: We had guys go out and not only do the internal usability testing you traditionally do, where you get a bunch of guys in and videotape them as they try to do some tasks on a machine. We also went out and spent time with real users, just sitting in and watching. And we learned a lot of stuff there, like what nine-tenths of the world finds very difficult. It turns out that nine-tenths of the world can't find their windows, nine-tenths of the world finds overlapping windows confusing. Most people run with their windows maximized all the time. . .

BS: . . .or only run single applications. These are common problems people have that we went out to solve, and one of the things we learned as we worked on the Chicago user interface is that by having a really good design you solve a lot of problems you never anticipated you were setting out to solve. Good design really means that you have a small

number of really good principles that work together, that combine freely and combine well. So that while we started out to make Chicago easy for novices, we found that having a small number of really useful, easy to combine principles means that we made the product a lot easier and a lot more powerful for power users too. That's the benefit of good design. I think we make quite a number of innovations and contributions in the Chicago user interface. I fully expect the developers of other operating systems to follow suit with some variation of what we're doing. Things like the taskbar. The taskbar is a breakthrough in how you manage multiple applications. On whatever graphical operating system, we've found that people can't do window management. They lose track of things. They don't run multiple applications because they just lose track of them. The taskbar makes it very easy for people to run multiple applications and not have to worry about window management. It's like Windows TV! You just click a button and you get the Excel channel or you get the Word channel or you get the Mail channel. It's a metaphor that people are very used to. It gives you an anchor point together with the Start button so that if you don't know how to get something done, you're led to that one place that's really the source of 90 percent of what the system can do.

The Start button. Having a uniform namespace so that all system objects are in a single namespace, so you don't have a Font Manager and a Program Manager and a File Manager and all these other managers. If you want to look at your printers, you go to the Printers folder. If you want to look at the attributes of your printer, you look at Properties on your Printers. You don't have to say, "I want to add a printer. Do I go to the Control Panel for Printers, do I want Setup or do I want something else. . ."

PM: Going back to your original question. People who say that this thing is like the Workplace shell, or Motif, or something else just really haven't used either product, or they wouldn't be able to say that.

BS: So having properties on all objects in the system—that's uniform. Anytime you see something, you know it has properties, and you can right-click and get to the properties. That eliminates the complexity bomb that would otherwise be there. If you want to add more and more capabilities to a system, unless you have this common framework that allows you to add things in a uniform way, you're just adding idiosyncratic feature after idiosyncratic feature. So the right-click for properties, the

taskbar and the Start button, shortcuts or links—whatever we end up calling them—I think will all be important. They change the way you work. They absolutely change the way you use the system. You never have to remember crazy pathnames all over the network anymore. You just create a folder. Single-click to close. Stupid little things, but once you get used to it and then you go back to 3.1, you say, "This is really awkward. How did we ever live with this?"

AK: *So coming from that, name your three favorite Chicago features.*

BS: The shell itself. For sure, just the whole look and feel and gestalt of the shell. Second, I love shortcuts. I think shortcuts, particularly shortcuts to network resources, change the way I use the product. They make me more efficient on a day-to-day basis. The third feature I'd say is the integration of the network. How the network is seamlessly integrated into the system.

PM: I think a lot of the Plug and Play features are pretty nice. And not just at the "stick the boards in and pull them out" level. It's the whole way you can go in and reconfigure your desktop without rebooting your system and having to dink around like that.

BS: Plug in a CD ROM and not have to spend the weekend doing it.

PM: I think a lot of the mobile features are pretty nice. It's a real nice system to take on the road on your laptop. There was a bunch of stuff in M5 *[the release distributed at the December 1993 developer conference—Ed.]* that we got cleaned up in Beta-1, and more still needs to be done, but you can see that it's going to be a lot better for mobile users. The Briefcase and all those kinds of features that are really cool. Thirdly, there are elements in the user interface that you think, Boy, how did we live without these things? Like the Document list and the Start button. You notice how much easier it is than if you have to open up the File Manager, find the directory, scroll down the directory list, and find the document and then open it. It cuts four or five clicks out of every operation. You realize you're getting to stuff far more quickly than you were before.

AK: *Do you think Chicago is MS-DOS 7.0? Or is there going to be a different animal called MS-DOS 7.0?*

PM: I think that for all intents and purposes Chicago is MS-DOS 7.0, if by that you mean that MS-DOS 7.0 is the next version of the software that every PC comes equipped with. Will there be a nongraphical product that will have the familiar C:\ prompt as its fundamental interface? And as such is it MS-DOS 7.0? It's an interesting question. You have to ask yourself, "What is the market for the end product?" There would have to be somebody who for some reason has a complete aversion to graphical user interfaces and refuses to use one under any circumstances. On the other hand, we've always been surprised by the number of people who want to buy an upgrade to MS-DOS.

AK: *Have you identified the people who like the C:\ prompt, or are you just guessing that they're out there?*

PM: That's why we haven't made a decision one way or the other whether we want to do MS-DOS 7.0. It's hard for us to figure out how many of these things we'd sell. Logic would say you're not going to sell that many.

BS: Chicago would run the same MS-DOS apps that such a product would. We put a lot of effort into our support for MS-DOS applications so that we could run anything that's out there. It's not as if an MS-DOS 7.0 would run applications that Chicago wouldn't. It just wouldn't be able to run Windows applications. We just don't know yet if there's sufficient demand. If there's enough demand, we'll build it.

AK: *When do you see the release of a fully Chicago compatible version of Windows NT happening? By that I mean a release with the new shell, Plug and Play, and all the rest of it.*

PM: That's the next release after Daytona, called Cairo. Our goal is to get that out during 1995.

AK: *Do you worry that people will simply dismiss Windows NT when Chicago hits the streets with all the attendant publicity? That they'll just sort of forget about it and assume that Microsoft has aced itself again?*

PM: There's a very real reason they won't forget about NT. NT is our offering, quite apart from any other issues, for the server market. So we'll continue to sell NT very aggressively in the server market, where it

offers tremendous advantages—where it can handle multiprocessors and offer security, reliability, and robustness—those sorts of things. Those features are not just "nice"—they're absolutely necessary.

And there are significant customers who have already selected Windows NT as their desktop operating system. They'll be buying Windows NT in fairly large numbers during 1995. These are customers like financial trading houses, who have long development and deployment cycles because they're planning to run some very critical applications. So there will be significant customers buying and deploying Windows NT during 1995. And our focus will be on servicing those customers. Windows NT is not an operating system that we have ever expected to sell through the corner store. It was built expressly in order to solve specific problems for people, and we'll concentrate our marketing efforts on servicing those customers. And then, when we get to Cairo, which does pick up the Chicago UI, that's when we'll expand our marketing of the NT product to an even broader segment of the corporate market.

AK: *Do you lose any sleep over the people who are trying to compete with you by attacking Windows? The WABI initiative, Taligent, OS/2, etc., etc.?*

PM: Do we take competitors seriously? Yes. We have to because of the very large sums of money that people are spending to compete with us. And these are not incompetent people, not stupid people. These are people who are very serious and have us steadily in their sights. We can't afford to grow lax or to ignore them. On the other hand, I think if we execute, if we deliver in a reasonable way, and above all, if we deliver quality, we'll be OK. My biggest concern with Chicago is that because it has to sell to so many people and be a successful upgrade for so many satisfied users today, it has to be a very high quality product. So if we execute well in a reasonably timely way and deliver a quality product, I think it's going to be a tough job for our competitors to try and match that.

AK: *Do you think it's technically feasible for somebody to run a Chicago-compatible system hosted on top of another operating system?*

BS: It's only software.

PM: It's a question of time. . .

AK: *Within our lifetime?*

PM: . . .and resources. You understand, we're not religious about this. We have licensed the Windows source code including the Chicago source code to people so that they can do precisely that—in the UNIX environment, for instance.

BS: If IBM wants to license Chicago, we're glad to license it to them. To us it's just a business decision. It's not a religious decision.

PM: Cloning these modern pieces of software is a tough challenge. I don't know the exact line count of Chicago, but it's millions of lines of code, and compatibility is just an incredible, incredible challenge. We have full access to all the Windows 3.1 source code and our test suites, and getting both Chicago and Windows NT to be compatible with Windows 3.1 and run all those applications has got to be the largest part, by far, of our expenditure of effort.

BS: All things said, I'd rather be playing our hand than their hand. We've got a tough challenge, and if we execute, we're in good shape. I'd rather be in our position than theirs.

AK: *You're re-emphasizing OLE with Chicago by including it as a standard component. How do you feel OLE is doing in terms of both the number of ISVs who are really adopting it and its position in competition with the other object architectures?*

PM: There's a tremendous amount of heat and light about "things object" at the moment—most of which has nothing to do with the average end user. This is truly an industry-induced storm here, where we're just talking to each other. But OLE is the only thing (a) that an ISV can concretely do something about and (b) that an end user can actually use to get some benefits from component-oriented software. We have done a lot of thinking about OLE, and a lot of design work has gone into it. A lot of what you hear bandied about, that OLE isn't good with a distributed environment, or isn't able to handle nonrectangular Windows, is all just nonsense. All that stuff has been thought about and provision made for it and, in fact, if you take the distributed case, designed very

elegantly for in the sense that all of the components that are written today will be able to play in a distributed environment with no change whatsoever. This is not true of models like DSOM, where you have to make source code changes to get your components to work in a distributed environment.

In terms of acceptance in the marketplace, the thing to do is to watch people's feet, not their mouths. There isn't any major software vendor who isn't making significant investments in OLE technology. OLE is a very broad thing. It's really an umbrella for a series of technologies—application automation, compound document support, etc. Not all ISVs are using all the options under that framework, but that's to be expected. It's like an operating system: not all ISVs use all the APIs in the operating system. There are many people making their applications OLE enabled. There isn't anybody of note at the moment who isn't.

AK: *The recent* Microsoft Developer Network News *listed "the magnificent seven" requirements for an ISV who wants to license the new Windows logo for display on the product box. One of these was that you've got to support OLE. That's a little bit aggressive, I would say. Why did you decide to do that?*

BS: I think to build a quality Chicago application requires developing Win32 OLE applications. That's part of what it means to build a great Chicago application.

PM: People should have certain expectations of their applications when they see that logo. What we're saying is that they should be able to see that this application, by virtue of carrying the logo, is going to be a first-class citizen in this environment. And, in our opinion, to be a first-class citizen this is what you need to do.

BS: Win32, OLE, long filenames. . .

PM: People don't have to use the logo. This is an issue of what you want the end user to be able to expect when he sees an application that has the Chicago logo on it.

AK: *One of the things I didn't understand looking through that requirements list was that a qualifying app must be able to run on Windows NT version 3.5. Given that you don't have all the Chicago facilities in that release, how does an ISV do that? On the one hand, you're insisting on adoption of the new look and feel, and on the other you're insisting on being able to run on Windows NT.*

PM: The answer is that we've made it very easy for people to produce a high-quality, first-class-citizen Chicago application and also have that application run on Windows NT 3.5. The controls that you'd use to get that new look and feel will be available on the Daytona platform, so we feel that that is actually a very modest requirement. And most ISVs plan to meet it.

AK: *So that will be a library that's going to ship with Daytona or a compiler or something?*

PM: Yes, with Daytona.

BS: The main thing that Daytona won't have will be integration with the shell. But that's OK because the key message for ISVs is that they just write to Windows. And there are two different implementations of Windows. There's the high-end NT implementation and there's the high-volume Chicago implementation. But it's just like when you write an Intel program: you don't write to a Pentium, you don't write to a 486, you just write to the Intel instruction set and depend on Intel to get the semantics of that instruction set uniform across the implementations. The same is true with Windows. We just want ISVs to write to Windows and leave it to Microsoft, with some testing by the ISVs, to make sure that it will run across the various implementations of Windows.

PM: And there are some rules you have to follow to do that, but by and large we feel that those are fairly commonsensical and that they won't be a big overhead.

AK: *Can you give some idea of the scope of the project? Number of programmers, testers, and those sorts of metrics.*

BS: I can't tell you exactly how many people. Chicago is done by my core team as well as by people both within Microsoft and outside Microsoft working on some external components. The OLE code, for instance, is done by a group in Daytona. Mail is done by a group in the Business Systems Division. And some components came from outside the company, like the file viewers, the terminal application, and the backup application. And I have no idea how many people are working on those components. If you eliminate those people, just within the

Chicago core group, it's approximately 350 people. That includes developers, program managers, testers, and marketing people. Of which, say 160 developers—I think there are 160 developers in the Chicago group. That's again just my team. That doesn't include Mail or OLE or some of the external components. And approximately the same number of testers.

AK: *Do you know the numbers of tests that have been done?*

BS: I know that to this point, we've done over 400,000 hours of stress tests. We've got about 20,000 beta sites. The product has been in a PDK (Programmers Development Kit) release for almost a year now. The first PDK was in August 1993. By the time we ship, it will be the most stress-tested, most beta-tested, most analyzed, most speculated-on piece of code ever delivered in the history of software. I think it's about 4 million lines of code altogether.

AK: *Do you think there are any features that you might yet drop?*

BS: Oh yes. I don't really want to discuss what they might be. But we have a list of features in the category "if we have a hard time with these, we'll find a way to get them done," and we've got another list of features in the category "if we have a hard time with these, they'll catch the next train." But as you can see from Beta-1, the product is awfully complete. In many ways, if we hadn't spent so much time talking about some of the features yet to come, it'd be a fine product—even if we didn't add anything that wasn't in Beta-1. We feel real good about the content that's in the beta. And stuff that's not yet in the beta? We hope to get most of it in, but if we don't, I'll still feel good.

AK: *And you're planning two more beta cycles before shipping?*

BS: Yes.

AK: *I think the first one went to about 20,000 people?*

BS: Beta-1 has gone out to about 15,000 now, and by the time we finish rolling it out it will be up to about 20,000.

AK: *Is that going to increase?*

BS: Oh yes, it'll only increase. And the last one will be truly massive. I mean, some of the numbers we're talking about are 100,000, 200,000. Because we want to make sure that the product really has those road miles underneath it so that when it comes out, people are really comfortable that it's solid production quality and they can roll it out broadly.

AK: *How many national languages are you going to ship in?*

BS: Simultaneously we will have seven languages. We'll go up to something like twenty-six languages altogether. And they will all be done within the first 180 days of shipment. The vast majority will come out within the first 30 to 60 days. The first seven languages are English, German, French, Italian, Swedish, Dutch, and Spanish.

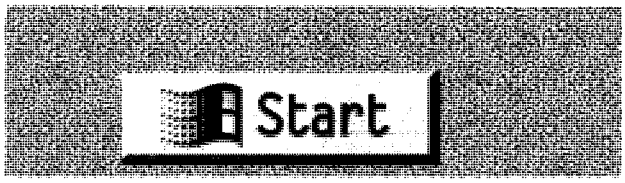
Let me give you an example of just how broadly we're going to localize Chicago. We're doing a Thai version. We just approved, this week, a Slovenian version of Chicago. We're doing a Catalan version of Chicago. We're doing a Basque version. So there's really nowhere in the world you can go and not be able to get a localized version.

AK: *. . .and not run into Chicago. And one final detail question. The Pen extensions were heavily emphasized early on in some of the product presentations, and then discussion of them kind of disappeared. What happened there?*

BS: They're in the product. We're definitely planning to include the Pen extensions with Chicago. The level of visibility they get, I think, will be commensurate with the level of visibility that pen-based machines will have in the market. A couple of years ago, they were getting a lot more visibility than they are now. Some pen-based products came out, but they weren't particularly successful. We still think there's a place for them, particularly in vertical markets. We're just building the Pen extensions in as part of the product. It's not worth calling out that much attention to them, but if companies are building pen-based machines, they'll know that the pen support will be there.

AK: *Thanks for all the information. Good luck with getting the product out the door.*

And there it is—Chicago circa July 22, 1994. No doubt the long road from Redmond has a few twists and turns yet to be revealed. I'm sure we'll all be watching with a great deal of interest.



GLOSSARY

0:32 addressing Memory addressing that uses the least significant 32 bits of the full address.

16:16 addressing Memory addressing that uses a 16-bit selector and a 16-bit address.

access control list (ACL) The data defining the access rights of network users to a particular network *resource.*

account See *user account.*

address book A database used by the *messaging* system to record usernames and electronic address information.

address space See *virtual address space.*

AEP See *asynchronous event packet.*

alias At one time, a synonym for *shortcut.*

API See *application programming interface.*

application programming interface (API) The defined set of functions provided by the operating system for use by an application.

appy time (application time) A Windows system condition in which it is safe for a *VxD* to make *filesystem* calls or request memory allocation services much as if it were an application program.

asynchronous event packet (AEP) A data structure used in the *filesystem* software to notify the lower layers of the occurrence of an event such as the completion of a data transfer.

asynchronous event routine A function that can be called by the operating system *kernel* upon the occurrence of a set of predefined events.

At Work Microsoft's office product automation initiative, designed to allow common devices such as photocopiers, facsimile machines, and personal computers to exchange information in a common digital format.

authentication Validation of a user's network logon information. See also *pass through authentication.*

automation See *OLE automation.*

base system The operating system components of Windows 95, comprising the memory management, *task* management, and *interrupt* management functions of the operating system.

Bézier curve A mathematical technique for drawing a curved path given a set of discrete points. Frequently used in computer-based drawing systems.

BIOS (and Plug and Play BIOS) The Basic Input Output System of the PC. The BIOS comprises the lowest-level interface to common devices such as the system clock, the hard disk, and the display. A *Plug and Play* BIOS supplements the BIOS functions with routines that support Plug and Play operations such as device enumeration.

bit blt A bit block transfer, an operation that moves a collection of bits from one place to another. The most common example is the transfer of an in-memory image to a display device.

block devices Devices addressed in terms of blocks of bytes, such as disks and tapes, as opposed to devices addressed in terms of single characters or *pixels,* such as printers or displays.

boot loader The software responsible for starting the operating system—typically after power on. In Windows 95, the boot loader is a modified form of MS-DOS.

briefcase A specialized shell folder that allows the synchronization of different versions of the same file.

browsing Looking around the network—locating files, programs, printers, and so on. See also *Explorer.*

bus A device that plays a role in the control of at least one other device. In the hardware context, adapter cards plug into a bus. In the *Plug and Play* context, any device that provides resources is a bus.

cache A transient storage area in main memory used for data that might be needed again in a very short time frame—for example, the directory information associated with a *filesystem.* Intel processors also implement a hardware cache to retain copies of frequently accessed memory locations. Windows 95 implements a shared cache (under control of the *VCACHE VxD*) used for file and network access and paging.

Cairo The codename for Microsoft's future release of the *Windows NT* operating system. See also *object filesystem.*

calldown chain An implementation technique (used in the *filesystem* architecture) that allows an arbitrary number of functions to be chained together for execution.

call gate See *gate.*

CDFS The Windows 95 *protected mode* implementation of an ISO 9660–compliant CD ROM filesystem.

CISC processor A complex instruction set computer processor. A CISC processor uses a large number of instructions containing multiple fields, addressing modes, and operands. Many CISC instructions take more than a single clock cycle to decode and execute.

client Usually a system attached to a network that accesses shared network *resources.*

client application A program that makes requests of a *server application* using a defined interface such as *named pipes, RPC,* or *NetBIOS.*

client-server networking A network architecture in which shared *resources* are concentrated on powerful *server* machines and the attached *desktop* systems fulfill the role of *clients,* making requests across the network for centralized information.

CMC See *Common Messaging Calls.*

CMOS memory Memory kept alive by the system battery. PCs use CMOS memory to store configuration information, and some *Plug and Play* systems use CMOS memory to store device information.

color profile The definition of a device's color capabilities and current calibration. Used by the *image color matching system.* See also *image color matching.*

COM See *Component Object Model.*

Common Messaging Calls (CMC) The set of calls defined by the X.400 API Association for use in messaging applications. Similar in scope to *Simple MAPI.*

Component Object Model (COM) The architecture from which *OLE* is derived. Microsoft is working to establish COM as an industrywide standard for object-oriented systems.

compound document An *OLE* term that describes a single document containing multiple data types and operated on by multiple *OLE server applications.* See also *container.*

compound file A file used by *OLE.* On Windows 95, a compound file is a single disk file that contains multiple independent data streams and indexing information.

configuration manager The component of the *Plug and Play* system that's responsible for managing the software configuration associated with a system's current hardware configuration.

connection A logical *link* between a local name and a network *resource.*

container In *OLE,* an *object* that can hold other objects. See also *compound document.*

contention A condition in which two or more active *threads* require access to a single *resource.* The operating system resolves the contention problem by providing a means for one *thread* to gain control of

the resource and thereby block access to all other threads. See also *mutual exclusion service (mutex)* and *semaphore.*

context menu See *popup menu.*

control A fundamental *object* in Windows that defines the appearance and behavior of a particular visual element such as a menu or a scroll bar.

cooperative multitasking An operating system scheduling technique that relies on running applications to *yield* control of the processor to the operating system at regular intervals. See also *preemptive multitasking.*

coordinate system The Windows *GDI* definition of the drawing space available to an application. The coordinate system follows the simple geometric model you learned in grade school.

critical section A sequence of instructions that must be guaranteed to execute without yielding control of the processor to another *thread.* A critical section is typically used to guarantee the integrity of a change to an in-memory data structure.

DC See *device context.*

DCB See *device control block.*

DDE See *dynamic data exchange.*

demand paging A technique that brings the memory pages of an application or operating system component into memory from disk only at the time the pages are needed. This technique is opposed to the one in which the entire memory image of an application is loaded when the application first starts. Demand paging requires support from the processor. Intel 386 and later processors provide this support. The earlier processors do not.

descriptor On the Intel 386 series processors, an 8-byte area of memory used to fully describe a region of memory. Descriptors are grouped into either a local descriptor table (LDT) private to the process, or a global descriptor table (GDT) shareable among processes.

Every address generated on the 386 includes a selector that identifies which descriptor table to use and includes the index of the descriptor in the table. The descriptor tables themselves are held in memory with special purpose processor registers used to hold the starting addresses of the tables.

descriptor table See *descriptor.*

desktop What you see on your Windows screen. Also the logical container managed by the *shell.* See also *Z order.*

despooler The system component responsible for taking the data in spool files and handing it to the software responsible for writing it to an output device.

device context (DC) A *GDI* data structure that describes the current state of a device or drawing surface.

device control block (DCB) A data structure used in the *IOS* to retain information about a particular hardware device.

device driver A generic term used to refer to the lowest-level software in an operating system that deals directly with the hardware of a particular device.

device-independent bitmap (DIB) An in-memory bitmap whose attributes are independent of any particular hardware device.

device node The logical *object* in the *Plug and Play* subsystem's *hardware tree* that is used to describe a specific device. Also called a *Plug and Play object.*

device virtualization A technique used in Windows to replicate the hardware characteristics of a device in a software interface. The virtualization technique allows more than one application to manipulate a single hardware device at the same time. The technique relies on hardware support from the Intel 386 processor. See also *VxD.*

dialog A visual element of Windows that groups one or more controls. Usually employed to interact with the user.

DIB See *device-independent bitmap.*

display driver The Windows component responsible for manipulating the display hardware. See also *mini-driver.*

DLL See *dynamic link library.*

DL VxD See *dynaload VxD.*

DMA channel A hardware interface that allows a device to transfer information to and from main memory without interrupting the processor.

document-centric design A design technique that focuses the user on documents and the information therein rather than on the applications generating the data that combine to form the document.

domain A collection of network *servers* and *resources* in a logical grouping.

DPMI The DOS Protected Mode Interface. An older technique for allowing 32-bit *protected mode* programs to run under MS-DOS.

driver registration packet (DRP) An *IOS* data structure used to initialize the logical connection between IOS and a particular device driver.

DRP See *driver registration packet.*

dynaload VxD (DL VxD) A dynamically loaded *VxD*—loaded as needed by the operating system.

dynamic data exchange (DDE) An older form of data exchange between two or more cooperating application programs. Windows 95 aims to replace the use of DDE with *OLE* or *RPC.*

dynamic link library (DLL) A library of shared functions that applications link to at runtime as opposed to compile time. A single in-memory copy of the DLL satisfies requests from all callers.

EGA The Enhanced Graphics Adapter. Under Windows 95, no longer supported.

EISA The Extended Industry Standard Architecture. A *bus* design that allows 32-bit adapters and some automatic device recognition and configuration. EISA hasn't achieved the success expected for it. See also *ISA*.

embedding An *OLE* term for the inclusion of an *object* within a *container*. The data associated with the *object* actually resides in the *container*. See also *link*.

enumerate To list a set of related *objects*—for instance, all of a *server*'s resources.

event The occurrence of a condition that's of interest to one or more software components. The term is typically used to describe the internal manifestation of an action such as a mouse click.

event-driven program A programming technique in which the application is driven by events rather than by data. The event-driven model dominates modern personal computer operating systems.

exception An event that results from an error such as division by zero. See also *structured exception handling*.

Explorer The *shell* function that provides the user with the ability to *browse* files, *folders*, and other *resources*.

export table The definition of callable functions included in a *DLL*. The linkage between an application and a DLL is formed by means of the entries in the export table.

Extended MAPI The complete set of Microsoft's *MAPI* functions. Extended MAPI enhances *Simple MAPI* by adding features such as *address book* manipulation and *message store* querying. See also *MAPI* and *Simple MAPI*.

FAT The File Access Table. The default MS-DOS *filesystem* organization.

filesystem A logical structure of files and associated indexing information, typically stored on a disk.

filesystem driver (FSD) The component of *IOS* that implements the interface to a particular type of *filesystem*. Windows 95 supports multiple concurrent FSDs.

folder A logical container implemented by the *shell* that allows the user to group any collection of items—a set of documents, for instance. Folders are most usefully thought of as directories.

frame buffer The region of memory directly associated with a display. Changes to the data in the frame buffer result in changes on the visible screen.

FSD See *filesystem driver*.

gate A specialized *descriptor table* entry that allows control transfers between *protection rings* on the Intel 386 processor.

GDI Graphics Device Interface. The component of Windows responsible for implementing the graphical functions such as line drawing and color management. GDI is a *DLL* that includes all of the graphical *APIs* in Windows.

GDT See *descriptor*.

geometry (of a device) The organization of a device, such as the number of sectors per track and bytes per sector of a disk drive device.

global descriptor table (GDT) See *descriptor*.

grabber See *screen grabber*.

granularity (of allocation) The amount of the smallest storage increment that can be used to satisfy any request for additional storage.

handle A program data *object* that provides access to an allocated Windows *resource*. Almost every item manipulated by a Windows application is addressed by means of a handle. Individual windows, memory regions, files, timers, and other *objects* have handles.

hardware tree The logical representation of a system's current hardware configuration built and managed by the *Plug and Play* subsystem.

heap A region of in-memory storage that can contain data items of different sizes, types, and attributes.

ICM See *image color matching.*

IFS See *installable filesystem.*

IFS manager See *installable filesystem manager.*

image color matching (ICM) A new Windows 95 subsystem responsible for the manipulation of color information in a way that is device-independent.

import library A compile time library used to satisfy references to external functions that will ultimately be resolved at runtime by a *DLL.*

in-place activation In *OLE,* a technique whereby a user can make use of functions of a *server application* on a data *object* in situ within a document. In-place activation supersedes the more common current technique, in which the user sees the screen display change focus to another application.

in-place editing See *in-place activation.*

installable filesystem (IFS) A technique used by Windows 95 and *Windows NT* in which more than one active *filesystem* type is supported by the operating system. Windows 95 allows an IFS to be dynamically loaded. See also *installable filesystem manager.*

installable filesystem manager (IFS manager) The component that provides the interface between application requests and the specific *filesystem* addressed by an application function. The IFS manager routes *filesystem* requests to the appropriate *filesystem driver (FSD).*

interrupt A hardware signal that causes the processor to begin execution at a different address upon completion of the current instruction. A hardware device uses an interrupt to gain the attention of the operating system. See also *interrupt service routine.*

interrupt service routine (ISR) A sequence of instructions executed as a result of a hardware *interrupt.*

I/O packet (IOP) An *IOS* data structure that describes a single data transfer operation.

I/O port An addressable location on the Intel 386 processor to and from which hardware control information is read and written.

IOS See *I/O supervisor.*

I/O supervisor (IOS) The Windows 95 subsystem responsible for control of the attached *block devices.*

IPX/SPX Novell's lower-level network *protocol.*

IRQ The *interrupt* request level. Each hardware device raises an *interrupt* on a predetermined IRQ (numbered 0 through 15). The processor associates specific interrupts with different *interrupt service routines.*

ISA The Industry Standard Architecture. An acronym used to describe PCs compatible with IBM's original IBM PC AT design. See also *EISA.*

ISR See *interrupt service routine.*

kernel The core component of an operating system. The kernel is usually considered to include the lowest level of memory, *interrupt,* and process management functions.

Kernel The Windows memory management, process management, and file management functions.

LDT See *descriptor.*

least recently used (LRU) technique A memory management technique used to ensure that a page reclaimed for use is the "oldest" (least recently accessed) page in memory.

legacy Older hardware and software still in use. In the *Plug and Play* context, the installed base of device cards that don't conform to the *Plug and Play* standard.

linear addressing A memory addressing scheme that organizes memory so that incrementing an address pointer guarantees a valid pointer to the next byte in memory. See also *segmented addressing.*

link An *OLE* term for a reference within a *container* to an *object* whose data is maintained by another application. Also used in earlier versions of the *shell* for *shortcut.*

local descriptor table (LDT) See *descriptor.*

locale A Windows term that refers to the system's current international configuration, including the national language and other items such as date and time formats.

locality of reference A program pattern of behavior that results in heavy access to closely grouped memory locations.

look and feel The appearance of a system and the response of the system to user input.

LRU See *least recently used technique.*

MAC driver See *media access control driver.*

MAPI The messaging *API* defined by Microsoft to allow applications to use a consistent interface to message-related subsystems such as those handling electronic mail messages, voice mail, and facsimile data. MAPI comes in two forms: simple and extended. See also *Extended MAPI* and *Simple MAPI.*

mapped file A file whose contents are directly addressable as part of an application's address space.

MDI The multiple document interface. A user interface technique that allows an application to support several active documents whose windows are clipped to the application's parent window. Microsoft is advising developers to discontinue use of MDI. See also *SDI.*

media access control driver (MAC driver) A device driver responsible for the lowest level of network device control. A MAC driver deals directly with the network adapter.

memory mapped device A device, such as a display, that can be addressed directly as part of the system's address space.

message In Windows, a message is a unit of data the operating system hands to an application to inform it of an event. The word *message* is also used as a generic term to describe the data manipulated by *MAPI*-based applications.

message loop The common Windows application program structure in which a control loop repeatedly receives and processes *messages*.

message store The structured storage associated with *messages* handled by *MAPI*-based applications.

messaging The generic term applied to applications that manipulate communicated information such as that found in electronic mail or voice mail messages, or facsimile documents.

metafile A file format that describes a series of graphical operations in a high-level, device-independent data format.

Micro Channel IBM's PS/2 series hardware *bus*.

mini-driver The hardware-dependent component of a device driver in which the driver is structured as a collection of shared functions and a smaller hardware-dependent driver module. Mini-drivers emerged first for printers and in Windows 95 are available for displays, modems, disks, and pointing devices. See also *universal driver*.

miniport driver In the Windows 95 filesystem architecture, a driver specific to a particular *SCSI* device.

monitor A low-level device driver responsible for interfacing to a printer, either directly or via the network. The monitor is specialized in that it can receive input from a (usually) output only device and, as a result, return status and error information to higher layers of the operating system.

MPR See *multiple provider router*.

multiple provider router (MPR) The routing component for Windows 95 network operations. The MPR, a 32-bit *protected mode DLL*,

implements network operations common to all network types. See also *print request router.*

multitasking An operating system feature that allows several independent programs to run concurrently.

mutex See *mutual exclusion service.*

mutual exclusion service (mutex) A software technique designed to ensure that only one *thread* can execute a certain sequence of instructions or gain the ability to manipulate a particular data structure, at one time. See also *critical section* and *semaphore.*

named pipe A high-level data exchange *protocol* used by *client-server* applications on Microsoft networks.

native mode The 32-bit mode of the 80386 processor.

NDIS See *Network Driver Interface Specification.*

NetBEUI transport The NetBIOS Extended User Interface. A *network transport* commonly used on Microsoft networks.

NetBIOS A high-level network interface that provides reliable, error-free transmission of data between two cooperating applications on a local area network.

Network Driver Interface Specification (NDIS) A software specification that defines the interaction between a *network transport* and the underlying *device driver.* The NDIS is vendor independent.

network filesystem driver A 32-bit protected mode *VxD* responsible for implementing the semantics of a particular remote *filesystem.*

network provider (NP) An implementation of the network service provider interface. Called by the *multiple provider router (MPR)* only, never directly by an application, the NP encompasses operations such as making and breaking network connections and returning network status information.

network transport The lowest layer of the network subsystem, responsible for transmitting and receiving data packets via the underlying network device driver.

not-present interrupt A fault condition generated by the Intel 386 to signify that a memory page is not currently present in main memory. See also *demand paging.*

NP See *network provider.*

object In formal terms, an encapsulation of both data and access methods, some or all of which may be usable by another application. Object-oriented techniques allow an object's developer to expose well-defined interfaces to the object's behavior and to hide the details of the object's implementation, which ought to allow the use of the object by many unrelated applications. Although the term is heavily used throughout Windows 95, in many cases it is simply a more attractive way of saying "data" or "thing." *Object* is also the current favorite for most overused term in the software industry.

object filesystem A *filesystem* designed by means of object-oriented methods and suitable for use by object-oriented applications. *Cairo* is reputed to have such a filesystem. *OLE compound files* are a prototype for an object filesystem.

ODBC Open Database Connectivity. Microsoft's standard for allowing applications to access different database systems by means of a common *API.*

OLE Microsoft's implementation of its *Component Object Model (COM)* architecture on Windows systems.

OLE automation A technique that enables a *client application* to control an *OLE server* without direct input from the user. The automation capability relies on an application's providing defined interfaces to its functions for use by the *client application.*

Open Datalink Interface (ODI) Novell's network device driver interface standard.

page On the Intel 386, a contiguous physical memory region of 4K.

paging See *demand paging.*

paragraph Originally a region of 16 bytes of memory on an Intel processor. It's becoming an obsolete term now that 32-bit linear addressing is here.

pass through authentication An *authentication* technique that relies on another system or software subsystem to perform validation. The caller-supplied information is passed to the validating system, and the results are passed back to the caller.

path In *GDI,* a description of a series of points that GDI can connect (the stroke) with a particular type of pen or brush. The characteristics of the pen determine the pattern and color (fill) of the connecting stroke. A path (or pathname) to a file or directory is a name that describes the logical location of the file or directory.

pathname See *path.*

PCI bus A *bus* definition whose design was led by Intel. The design is intended to support high-speed 32-bit data paths between devices, memory, and the processor. *Plug and Play* fully supports the PCI bus.

PCMCIA A *bus* definition that defines a hardware interface suitable for peripherals with a very small (credit card size) form factor. Such peripherals are typically used on portable machines, for which weight, size, and power consumption are important considerations.

peer-to-peer networking A network architecture in which each connected system can act as both *client* and *server.*

persistent connection A network connection that has a lifetime beyond a single session or working day. The Windows 95 *shell* will return persistent connections to their prior states when the user logs in to the network.

physical address A memory address whose physical location matches its address. See also *virtual address.*

pixel The smallest element of a display that can be modified under software control. Pixels typically have color attributes individually associated with them.

Plug and Play The specification for a hardware and software architecture that allows automatic device identification and configuration. In Windows 95, the *Plug and Play* subsystem is responsible for these functions on behalf of the operating system.

popup menu A menu that appears disconnected from other visual elements (unlike the drop-down menus associated with most application menu bars). Windows 95 frequently displays popup menus when the user clicks the right (secondary) mouse button. Popup menus are sometimes called shortcut menus or context menus.

port driver A component in the Windows 95 *filesystem* architecture that controls a specific adapter. A port driver manages adapter initialization and device *interrupts.*

POSIX A definition of a standardized UNIX. The POSIX standard is not supported by Windows 95.

PPP The point to point *protocol.* An industry standard protocol intended for use over lower-speed, potentially unreliable connections such as telephone lines.

preemptive multitasking An operating system scheduling technique that allows the operating system to take control of the processor at any instant regardless of the state of the currently running application. Preemption guarantees better response to the user and higher data throughput. See also *scheduler.*

print request router (PRR) The routing component for Windows 95 print requests. The application calls are directed to the appropriate print subsystem via the PRR.

process A common term, used also by Windows 95, to describe the running state of a program.

property An attribute of an *object.* The term is used widely throughout Windows 95 to describe settings such as the color of a title bar or the

connected state of a modem. The guidelines for Windows 95 applications suggest that an *object*'s properties should always be available as the result of a right mouse click. See also *property sheet.*

property sheet A new Windows 95 dialog box intended to allow the convenient grouping of an *object*'s *properties* in a single place.

protected mode A mode of the Intel 386 processor in which the hardware carries out numerous validation checks on memory references, function calls, *I/O port* accesses, and other items. A protection failure allows the operating system to gain control and deal with the condition. An application must run in *protected mode* if it is to make use of the full address space and *virtual memory* capabilities of the 386.

protected mode mapper In the Windows 95 *filesystem* architecture, a module that disguises *real mode* drivers so that new *protected mode filesystem* modules don't have to take account of the different interface for existing MS-DOS drivers.

protection ring One component of the Intel 386 processor's *protected mode* validation capabilities. Windows 95 uses protection ring three for application-level software and ring zero for operating system components. Software executing at ring three can be prevented from executing privileged instructions or accessing defined memory regions. Software executing at ring zero has no such restrictions placed on it.

protocol The definition of an interaction between two software components that ensures reliable, error-free communication between the components. Typically used to refer to network-based exchanges.

protocol stack The collection of software modules that implement a particular network *protocol.*

PRR See *print request router.*

RAS See *remote access services.*

rasterizer The software component that turns a description of a font into a physical rendition of the characters suitable for use on a display or a printer device.

raw input queue The data structure maintained by the operating system into which all input *events,* such as mouse clicks and keystrokes, are placed before they are distributed to the *message* queues associated with individual applications.

real mode The Intel 8086–compatible mode of the Intel 386 processor. Real mode allows no access to the 386's large *virtual address space* or *demand paging* capabilities. Real mode does not enable the processor's protection system.

real mode driver An existing MS-DOS device driver that Windows 95 will run in *virtual 8086 mode.*

redirector The *client*-side software that accepts file access requests and transforms them into network requests.

registry A database maintained by Windows 95 for storing hardware and software configuration information. The registry is used heavily by the *Plug and Play* subsystem.

remote access services (RAS) A Windows 95 subsystem that implements remote dial-in and *connection* functions. See also *remote network access.*

remote network access (RNA) In Windows 95, the subsystem that allows a remote user to log in to a network much as if he or she were logging in locally. By means of RNA, network *resources* become accessible to the remote user.

remote procedure call (RPC) A software technique that allows an application to execute a function call in which the callee is executing on another machine on a network.

resource A network *object* such as a printer, or a collection of files grouped in a directory, that is available for shared access.

resource arbitrator A component of the *Plug and Play* system that understands the specific hardware *resource* requirements of a particular device and can resolve conflicts between devices that request the same *resource.* The arbitrator allocates the resources that will satisfy the device's requirements.

rich text Textual information that includes formatting information such as font, layout, and other *properties.*

ring See *protection ring.*

RISC processor A reduced instruction set computer processor. A RISC processor uses a small number of simple instructions. The technique allows the processor chip to be smaller (it has fewer transistors) and thus faster (the paths between individual gates are shorter), and cooler (so that it can run at higher clock speeds). Typically, every instruction on a RISC chip executes in a single clock cycle. See also *CISC processor.*

RNA See *remote network access.*

RPC See *remote procedure call.*

safe driver In Windows 95, a *real mode* driver whose functionality can be offered by an equivalent *protected mode* driver. The protected mode driver can thus take control of the real mode driver and safely bypass it while the system is running in protected mode.

scheduler The operating system component responsible for allocating processor time to a *thread* for execution.

screen grabber The component of a Windows display driver that saves and restores the screen state on behalf of an MS-DOS *virtual machine.*

SCSI The Small Computer System Interface. An industry standard hardware *bus.* SCSI devices respond to a defined set of commands and can be addressed by means of a unit number.

SCSI manager The Windows 95 *filesystem* component that provides the translation between a *Windows NT miniport driver* and Windows 95.

SDI The single document interface. SDI (in comparison to *MDI*) uses one window per document. Users switch between full screen windows (and thus documents) rather than switching between child windows within an application's parent window.

segment On the Intel 386, a region of *virtual memory* specified by a single *descriptor.*

segmented addressing An Intel processor memory addressing scheme in which the address is specified as the combination of a segment and an offset within a segment. This addressing technique (finally) goes the way of the dodo in use of the Win32 *API* on Windows 95. See also *linear addressing.*

semaphore A software mechanism used to implement *resource* or *critical section* management. A semaphore differs from a *mutex* in that it has a finite value that is usually greater than 1 initially. The controlling entity can thus allocate a predetermined number of copies of a particular *resource.*

server The system on a network that owns the *resources* available to *clients.* Server resources can be files, printers, or *server applications* (such as a multiuser database).

server application The software that controls access to a *resource* via a programmatic interface. *Client* software typically connects to a server application using one of the supported high-level *protocols* such as *named pipes* or *RPC.*

service provider A component of *WOSA* that provides the lower-level interface to a specific service, such as a messaging system, a database system, or a mainframe communications system. The Service Provider Interface (SPI) is defined for each service but never called directly by an application.

service table The definition of functions supported by a *VxD* and available to other VxDs.

shared memory A technique that allows a memory region to appear in the *virtual address space* of more than one *process.* Windows 95 supports a variety of shared memory features.

share-level security A network security method that relies on the administrator to associate access privileges with each network *resource.* See also *access control list.*

share name The name given to a *share point.*

share point A file *resource* that a remote user can connect to. All of the directories and files in the share point's subtree become part of the connected network resource.

shell A program that provides the user with a means of control over the system. In Windows 95, the shell controls the *desktop* and much of the interaction with the system's *resources.*

shell VxD The *VxD* responsible for loading the ring three components of the system. The shell VxD also implements services that allow *messages* to be sent between applications and VxDs.

shortcut A *shell* technique that allows the use of an alternative name to refer to an object. Many shortcuts can be defined for a single object. Shortcuts were at one time or another in the development of Windows 95 called *links* or aliases.

Simple MAPI The basic message addressing, transmission, and reception features of Microsoft's messaging *API* subsystem. See also *MAPI* and *Extended MAPI.*

SMB protocol The Server Message Block network *protocol.* The default protocol for Microsoft networks.

sockets The application interface to a *TCP/IP protocol* stack.

SPI See *service provider.*

spooler The component that takes application generated output intended for a printer and stores it temporarily on disk.

Start menu The name for the *shell*'s most obvious access point to the functions of Windows 95. The *popup menu* associated with the Start button on the *taskbar.*

static VxD A *VxD* loaded during the system boot process and never unloaded.

structured exception handling A software technique that enables controlled recovery from unexpected error conditions.

swap file The disk file used by Windows 95 to hold the active system and application memory pages that are not currently present in main memory.

system tray The early name for the Windows 95 *taskbar.*

system VM The *virtual machine* context in which all Windows applications execute.

TAPI The Telephony *API.* Microsoft's API definition for the *WOSA* telephony functions.

task Synonymous with *process.*

taskbar The final (?) name for the Windows 95 *shell* visual element that gives the user access to the *Start menu* and to currently running programs.

TCP/IP The Transmission Control Protocol/Internet Protocol. The default wide area network *protocol* used by both Windows 95 and *Windows NT.*

thread A single path of execution within a *process.* A single process can initiate multiple threads. The threads in a process share the code and global data of the parent.

thumbnail In *OLE,* the reduced image of a document stored within an *OLE compound file.* The *shell* can display OLE thumbnails to help the user during file *browsing* operations.

thunk An implementation technique that, for example, allows 16-bit code to call 32-bit code and vice versa. Originally defined simply as a piece of code that gets you from one place to another.

timeslice The amount of processor time the *scheduler* allocates among *threads* before its next evaluation of thread priorities.

transfer model The conceptual process of moving data from one application location to another. Implemented under Windows 95 using the Cut, Copy, and Paste operations.

transport See *network transport.*

TSD See *type specific driver.*

type specific driver (TSD) A component of *IOS* that manages all devices of a particular type.

UAE Unrecoverable Application Error. An error that would compromise the integrity of the system if it were to be ignored. In reality, it's a bug in the application program.

UNC See *Universal Naming Convention.*

Unicode A standard that defines an international character set encoding scheme.

Unimodem The Windows 95 name for the universal modem driver. In reality, a driver-level component that uses modem description files to control its interaction with the communications driver *VCOMM.*

universal driver A shared set of hardware-independent functions called on by the *mini-drivers.* Originally used by printer drivers, in Windows 95 used by modem, display, disk, and pointing device drivers.

Universal Naming Convention (UNC) A file naming convention that uses a \\NAME prefix to specify a network-unique path for a file or directory.

UNIX An operating system with many features similar to those of *Windows NT,* including *multitasking* and multithreading. Available on many different hardware architectures, with versions from Sun Microsystems, Novell, IBM, and others.

User The Windows 95 component that implements the window, *dialog,* and *control* manipulation capabilities of the system.

user account A database of information, accessed by means of the user's network logon name, that defines the user's access rights to network *resources.*

user level security A network security method that associates *resource* access privileges with a particular network login name.

VCACHE The *VxD* that implements a common disk caching capability used by all the *filesystem* drivers.

VCOMM The *VxD* that implements the common communications *port driver* functions.

vendor supplied driver (VSD) A layer in *IOS* that allows a particular vendor to extend *IOS* functionality.

VFAT The *protected mode* implementation of the *FAT* filesystem.

VFLATD The universal display driver *VxD*.

VGA Video Graphics Adapter. The default display type for Windows 95.

virtual 8086 mode The Intel 386 processor mode that allows an operating system to run software in an Intel 8086–compatible fashion while retaining a degree of protection.

virtual address An address in a thread's virtual address space. The physical memory corresponding to a particular virtual address may or may not be present in main memory. See also *demand paging, physical address,* and *virtual address space.*

virtual address space The collection of addresses that make up the total *virtual memory* allocated to a particular *thread.*

virtual machine (VM) The Windows context for execution of an application. The context includes a *virtual address space,* processor registers, and privileges.

virtual machine manager (VMM) The component of the Windows 95 base system that controls the initialization, *resource* allocation, and termination of individual *virtual machines.*

virtual memory Memory allocated to the address space of a *thread* but not necessarily present in main memory, or indeed not necessarily backed up by physical memory.

visual cue A technique used by the Windows 95 *shell* to suggest the purpose behind a particular visual element, or an association between different elements.

VM See *virtual machine.*

VMM See *virtual machine manager.*

volume tracking driver (VTD) The component of *IOS* responsible for managing removable devices.

VTD See *volume tracking driver.*

VxD Literally, virtual anything driver. A low-level software component that manages a single *resource,* such as a display screen or a serial port, on behalf of possibly many concurrent *threads.* This enables, for example, applications running in separate MS-DOS *VMs* to use a single screen. A VxD is always 32-bit *protected mode* code and is frequently written in assembly language.

widening The expansion of a bit quantity to a larger number of bits. Typically used to transform 16-bit integers into 32-bit integers of the same value.

Win16 The 16-bit subsystem of Windows 95.

Win16Lock The old name for *Win16Mutex.*

Win16Mutex The software *semaphore* that controls entry to the nonreentrant components of the 16-bit *kernel.* Called *Win16Lock* early on in the Windows 95 project.

Win32 The 32-bit subsystem of Windows 95.

Win32s The subset of the Win32 *API* implemented for Windows 3.1.

window menu What used to be called the system menu.

window procedure The function in a Windows application that is associated with a specific window.

Windows NT Microsoft's high-end 32-bit operating system.

Windows Open Services Architecture (WOSA) Microsoft's umbrella term for its definition of application-specific services, such as *MAPI* and *ODBC,* available under Windows.

Windows Sockets The Windows implementation of the *TCP/IP socket* interface.

working set The collection of memory *pages* belonging to a particular *thread* that must be present in main memory for the thread to execute.

WOSA See *Windows Open Services Architecture.*

yielding An application's handing control back to the operating system. See also *cooperative multitasking.*

Z order The order in which windows appear on the *desktop.*

INDEX

Numbers

0:32 addressing, 143, 427
3Com, 313
3-D appearance, shell, 184, 198–200
8-bit processor, 38
10Net program, 343
16:16 addressing, 143, 427
16-bit vs. 32-bit code
  calls and returns between, 144–47, 148, 149
  mixing, 54, 107, 142–47, 148, 149
  porting process, 229–33
  and process preemption, 149–55
16-bit Windows applications
  APIs for, 110–11
  further development status, 81
  message queue for, 120, 121
  running under Windows 95, 33–34, 64, 65, 81
  running under Windows NT, 33
  virtual address space, 25, 109
32-bit Windows applications. *See also* Win32 API
  message queue for, 120, 121
  and preemption, 26–27
  support for applications developers, 5–6, 24–27, 54
  and System VM, 71–72
  virtual address space, 25, 27, 85, 86–88, 109, 110, 125, 126
  and Win32 API, 224–38
  as Windows 95 component, 64, 65
  Windows 95 shell as, 147, 188
286. *See* 80286 processor
386. *See* 80386 processor
386 native mode, 37
486. *See* 80486 processor
640K barrier, 23, 39–40
8080 processor, 38
8086 processor
  1-megabyte memory limit, 38, 39
  compatibility with 80286 processor, 36, 37, 41
  compatibility with 80386 processor, 36, 37, 45, 68
  first introduced, 35
  memory architecture, 37, 38
  segmented addressing, 37, 38
8088 processor, 35
80286 processor
  compatibility with 8086 processor, 36, 37, 41
  deficiencies of, 36, 37, 43–44
  as faster 8086, 36, 37
  and IBM PC AT computers, 36
  as major architectural revision, 35, 36
  and MS-DOS 3.0, 36
  overview, 41–43
  in protected mode, 36, 41–43
  in real mode, 41
  segmented addressing, 41–43
80386 processor
  16-bit applications for, 54
  and 32-bit mode, 37, 44, 45
  benefits of, for MS-DOS–based applications, 59, 60–61
  compatibility with 8086, 36, 37, 45, 68
  compatibility with 80286, 36, 37, 41
  descriptor format, 45–48
  as major architectural revision, 35, 36, 44
  and memory addressing, 45–54
  Microsoft's role in developing, 44
  and need for new operating system, 44–45
  and operating modes, 45
  overview, 43–45
  privilege levels of, 56–57
  in protected mode, 45
  protection capabilities of, 54–60
  in real mode, 45
  and segment feature, 45, 46
  and software compatibility, 45
  successful follow-on to 286, 37, 44
  system performance, 1
  and virtual 8086 mode, 37, 44, 45, 68
  and Windows, 1, 37, 44–45
80486 processor, 35, 37, 44

A

access control lists (ACLs), 373, 378, 427
access controls. *See* access control lists (ACLs)
accessed bit, 47, 53
active users, defined, 30
adapter cards, 20, 315, 317

AddJob() API function, 236
AddObjectToBriefcase() API, 403
address book, defined, 427
addressing, 16-bit vs. 32-bit code, 25, 143
address registers, 38, 41, 42
address space. *See* virtual address space
Adobe Systems, 273
AEP. *See* asynchronous event packet
AEP_BOOT_COMPLETE message, 303
AEP_CONFIG_DCB message, 303
AEP_DEVICE_INQUIRY message, 303
AEP_INITIALIZE message, 303
AEP_IOP_TIMEOUT message, 303
alias, defined, 427
anchor point, taskbar as, 179, 180
animation, 196–97
ANSI character set, 235
appearance, screen. *See also* screen display
  of dialog boxes, 94, 164, 165, 211–13
  elements of, 198–213
  new controls, 205–10
  overall screen appearance, 22, 182–84, 198–201
  screen elements, 201–13
API. *See* application programming interface; Win32 API
API functions
  AddJob(), 236
  BroadcastSystemMessage(), 242
  CreateDC(), 270
  CreateDIBSection(), 266
  CreateDirectory(), 290
  CreateEnhMetaFile(), 272
  CreateEvent(), 240
  CreateFile(), 137, 138, 241, 290
  CreateFileMapping(), 127
  CreateMutex(), 240
  CreateSemaphore(), 240
  CreateWindow(), 149
  DeleteFile(), 290
  DeviceIoControl(), 137, 138, 139, 286
  DispatchMessage(), 97
  Dos3Call(), 233
  DuplicateHandle(), 240, 241
  EndDoc(), 270
  EndPage(), 270
  EnterCriticalSection(), 239
  Escape(), 269
  FindClose(), 290
API functions, *continued*
  FindFirstFile(), 290
  FindNextFile(), 290
  FreeImageColorMatcher(), 260
  GetBrush(), 148
  GetCurrentDirectory(), 290
  GetCurrentProcess(), 110
  GetCurrentTask(), 110
  GetFileAttributes(), 290
  GetFileTime(), 290
  GetLastError(), 233
  GetMessage(), 82, 97
  GetStockObject(), 148
  GetSysColor(), 245
  GetThreadDesktop(), 233
  GetVersion(), 233
  GetVolumeInformation(), 290
  GlobalMemoryStatus(), 122
  HeapCreate(), 129
  InitializeCriticalSection(), 239
  InterlockedDecrement(), 239
  InterlockedExchange(), 239
  InterlockedIncrement(), 239
  LeaveCriticalSection(), 239
  LoadImageColorMatcher(), 260
  MapViewOfFile(), 127, 128, 241
  MapViewOfFileEx(), 127, 128
  MessageBox(), 83
  MessageBoxEx(), 249
  MoveFile(), 290
  MsgWaitForMultipleObjects(), 240
  OpenFile(), 290
  OpenFileMapping(), 127
  OpenMutex(), 240
  OpenSemaphore(), 240
  PulseEvent(), 240
  RaiseException(), 252
  RasDial(), 389
  RasEnumConnections(), 389
  RasGetConnectStatus(), 389
  RasHangup(), 389
  ReleaseMutex(), 240
  ReleaseSemaphore(), 240
  RemoveDirectory(), 290
  ResetEvent(), 240
  ScheduleJob(), 236
  SetCurrentDirectory(), 290
  SetEvent(), 240
  SetFileAttributes(), 290

API functions, *continued*
SetFileTime(), 290
StartDoc(), 270
StartPage(), 270
VirtualAlloc(), 128–29
WaitForMultipleObjects(), 240
WaitForMultipleObjectsEx(), 240
WaitForSingleObject(), 240
WaitForSingleObjectEx(), 240
WNetAddConnection(), 358, 359
WNetAddConnection2(), 358, 359
WNetAuthenticationDialog(), 360
WNetCachePassword(), 360
WNetCancelConnection(), 359
WNetCancelConnection2(), 359
WNetCloseEnum(), 359
WNetConnectionDialog(), 359
WNetDeviceGetFreeDevice(), 360
WNetDeviceGetNumber(), 360
WNetDeviceGetString(), 360
WNetDisconnectDialog(), 359
WNetEnumResource(), 359
WNetGetConnection(), 359
WNetGetLastError(), 360
WNetGetSectionName(), 360
WNetNotifyRegister(), 359
WNetOpenEnum(), 359
WNSetLastError(), 360
WNetUNCGetItem(), 360
WNetUNCValidate(), 360
WriteProcessMemory(), 123, 241
Apple Macintosh, 339, 343
AppleTalk, 343
application developers
32-bit applications support, 5–6, 24–27, 54
adding OLE capability, 100, 217–18, 220, 245–48
developer relations group (DRG), 29
guidelines for, 217–20
and international support, 248–49
marketing Windows 95 to, 29
and memory management, 241
and multitasking, 238–41
and online help system, 186, 219
and Plug and Play subsystem, 241–42
and the registry, 242–44
and user interface, 245
Windows programming basics, 96–100
application platforms, 2, 5–6
application program errors, 2, 17, 56, 117–18
application programming interface (API), 142–55. *See also* Win32 API
and 32-bit support, 25
defined, 64, 65, 351, 427
functions, 71
Windows 95 Win32 API set, 26
applications. *See also* 16-bit Windows applications; 32-bit Windows applications; MS-DOS–based applications
backward compatibility with Intel chips, 36–37
common dialog boxes, 210–13, 217, 219, 227
compatibility with taskbar, 180–81
icons for, 204–5
memory management, Windows 95, 87–88
messages to, 119–21
OLE client vs. OLE servers, 246, 247
in Plug and Play systems, 323, 338–39
and privilege levels, 56
and protection rings, 108
starting, from Windows 95, 59
and UAEs, 56
Windows 95 base system support, 141–55
appy time (application time), defined, 427
architecture
of Intel processors, 37–45
of PCs, 4–5
segmented memory, 38–39, 41–43
Windows 95 filesystem, 277–81
Windows 95 GDI subsystem, 255–56
Windows 95 network subsystem, 347–55
Windows 95 printing subsystem, 269, 270, 272
Artisoft, 345
assembly language, thunks in, 144
asynchronous event packet (AEP), 301, 427
asynchronous event routine, defined, 428
At Work, defined, 428
authentication, 360, 366, 428
AUTOEXEC.BAT file, 21, 71, 73, 112, 242, 243

B

backward compatibility, 36–37
bank-switched video adapters, 267–68
Banyan networks, 27, 28
base address, descriptor table entry, 46
base system, Windows 95
application support, 141–55
components of, 66–67

base system, Windows 95, *continued*
defined, 66, 428
features of, 103–4
and privilege levels, 84
virtual device drivers (VxDs), 67, 84–85
Virtual Machine Manager (VMM), 67, 111–41
Beta-1 release, xxv, 30
Bézier curve drawing, 257, 428
BillG reviews, 191
BIOS, 4, 5
defined, 428
Plug and Play standard, 315, 317, 324, 336–37, 428
bit blt, defined, 428
bitmaps. *See* device-independent bitmap (DIB) engine; I/O permission bitmaps
block devices, 277, 428
boot loader, defined, 428
bottom line, 30
briefcase object, 400–404, 428
BroadcastSystemMessage() API function, 242
browsing
defined, 354, 429
design evolution of, 195–96
using Windows 95 shell, 169–70, 177
bus architecture, Plug and Play standard, 20, 315–17
bus devices, 329, 429
button list box control, 206
buttons
control of, 183
dialog box, 93
on system taskbar, 194, 195
window, 183, 204
byte granularity, 47

C

C++, 246, 247
cache, defined, 429
Cairo project
defined, 429
and document-centric interface, 159
as object-oriented system, 11, 166, 247
and OLE, 220, 246, 247–48
and RPC, 367–68
team involvement in Windows 95 shell design, 190, 191
and visual design issues, 166
vs. Windows 95, 6–13
as Windows NT version, 10
calldown chains, 297, 300–301, 429
call gates, 57
Call_Global_Event service, 132
Call_When_Idle service, 132
ccMail, 396
CDFS (CD ROM filesystem) driver, 280, 429
checkboxes, 93
Chicago project, xxv–xxvi, 1. *See also* Cairo project; Windows 95
child windows, 95
Chkdsk program, 24
CIENZY standard, 259
CISC processor, defined, 429
C language, and 16-bit vs. 32-bit code, 143
client applications, defined, 429
client machines
defined, 8, 429
multiple, Windows 95 support for, 28
requirements for, 8
and Windows 95, 9, 11
Windows 95 support for, 28
client-server networking, 8–10, 28, 341–42, 343, 344, 429
Clipbook, 197
Close button, 183, 204
CMC (Common Messaging Calls), 397, 430
CMOS memory, defined, 430
CMYK color standard, 260
color management systems, 259–62
color profiles, 260, 261, 430
color reproduction, 259–62
color scheme, changes in, 200
color space, 259, 261–62
column heading control, 207
COM (component object model), 247, 430
COMMAND.COM file, 159, 214, 282
COMMDLG.DLL file, 211
COMM.DRV module, 392
common dialogs
new visual style, 211–13
use of, 210–13, 217, 219
Win32 APIs, 227
Common Messaging Calls (CMC), 397, 430
communications. *See also* portable computers
COM ports, controlling, 57–60
port drivers, 392–94
Win32 APIs, 228
and Windows 95, 27–28, 60
Compaq, and Plug and Play standard, 4

compatibility
backward, 36–37
MS-DOS–based application issues, 4, 7, 44, 45
and NDIS (Network Driver Interface Specification), 369–70
network transport issues, 366–68
Plug and Play issues, 319–20
as Windows 95 requirement, 4, 7, 14–15, 44, 45
component object model (COM), 247, 430
COM ports, controlling, 57–60
compound documents, 167, 246, 247
defined, 246, 430
previewing, 212
compound files, 247, 430
CompuServe, 189
CONFIG.SYS file, 21, 71, 73, 112, 242, 243
configuration, hardware, and Plug and Play, 20, 315–17, 318
configuration files, Windows 95, 243, 332
configuration manager
defined, 322, 324, 430
role in Plug and Play, 141, 322, 323, 324, 333–34
console APIs, Windows 95 vs. Windows NT, 235
containers
on networks, defined, 354
in OLE, defined, 430
contention, in multitasking, 79–80, 430–31
context, 68, 70–71, 72, 73
context menus. *See* popup menus
continuation menus, 174, 175
control blocks, VM, 130
control objects
button list box, 206
column heading, 207
defined, 94, 95, 431
list view, 210
new, 205–10
progress indicator, 208
property sheet, 185–86, 209–10
rich text, 209
slider, 208
spin box, 208–9
status window, 206–7
tab, 209
tool bar, 205–6
tree view, 210
Win32 APIs, 227
Control Panel program, 22, 162
cooperative multitasking, 77–78, 431
coordinate systems
16-bit vs. 32-bit systems, 232, 235, 257
defined, 431
Windows 95 vs. Windows NT, 232, 235, 257
Cougar project, 104
CreateDC() API function, 270
CreateDIBSection() API function, 266
CreateDirectory() API function, 290
CreateEnhMetaFile() API function, 272
CreateEvent() API function, 240
CreateFile() API function, 137, 138, 241, 290
CreateFileMapping() API function, 127
CreateMutex() API function, 240
CreateSemaphore() API function, 240
CreateWindow() API function, 149
critical sections
defined, 239, 431
managing, 79–80
Win32 APIs, 239
Ctrl+Alt+Del, 135, 154
cursor, 268
customer benefits of Windows 95, 29
cut and paste operations, 197–98

D

data structures, Windows 95 use of, 238
DCBs (device control blocks), 300, 432
DC (device context), defined, 432
DDE (dynamic data exchange), 228, 245, 433
debug VMM services, 141
DEC Alpha processor, 33
default startup screen. *See also* screen display
design evolution, 192–93
Windows 95 vs. Windows 3.1, 157, 158
DeleteFile() API function, 290
demand paging, defined, 431
descriptor privilege level (DPL), 47
descriptors, 41, 42, 109, 431–32
descriptor tables, 41, 42, 45–48
desktop, Windows 95
animation on, 196–97
defined, 432
folders on, 172–73, 177
initial, 174–76
look and feel of, 177
overview, 174–79
Desktop dialog box (Windows 3.1), 164, 165

directory, 172
Desktop property sheet, 185, 186
despooler, 270, 432
DESQview, 37
developer relations groups (DRG), 29
developers. *See* application developers
device context (DC), defined, 432
device control blocks (DCBs), 300, 432
device data blocks (DDBs), 299
device drivers. *See also* mini-drivers
  and asynchronous events, 301
  controlling peripherals with, 4, 58–59
  defined, 432
  and device-independent capability, 253–54
  driver registration packets (DRPs), 298–99
  initialization, 298–99
  and IOS services, 298
  and MS-DOS–based applications, 58–59, 90
  and Plug and Play subsystem, 20, 324, 337–38
  protected mode, 24, 67
  real mode, 67, 69, 281, 307–8, 445
  virtual, 67
  virtualizing devices, 90–91
device identifiers, 331–32
device-independent bitmap (DIB) engine
  and bank-switched video adapters, 267–68
  defined, 432
  and display mini-driver, 254, 264, 265, 266, 268
  in GDI architecture, 254, 255, 256, 262, 263
  interfacing with, 268
  overview, 253
  and universal printer driver, 253–54, 272–73
DeviceIoControl() API function, 137, 138, 139, 286
device nodes, 329–31, 432
devices. *See* hardware
device virtualization, 60, 90–91, 432
dialog boxes
  appearance of, 94, 164–65, 211–13
  common, 210–13, 217, 219, 227
  controls in, 94, 95
  defined, 432
  elements in, 93–95
DIB engine. *See* device-independent bitmap (DIB) engine
DIBENGINE data structure, 266, 267
directories, 171, 172–73. *See also* folders
dirty bit, 53
DispatchMessage() API function, 97
display drivers
  defined, 433
  DIB engine/mini-driver combination, 254, 264, 265, 266, 268
display screen. *See* screen display
display subsystem
  and DIB engine, 265–66
  and display mini-driver, 254, 264, 265, 266, 268
  overview, 262–64
DLLs (dynamic link libraries), 82, 147, 433
DL VxDs, 133, 433
DMA channel, defined, 433
document-centric interface, 159, 166–67, 433
domains, network, defined, 354, 433
DOS386.EXE file, 129
Dos3Call() API function, 233
DOS extenders, 69, 74
DOS Protected Mode Interface (DPMI) specification, 24, 74–75, 433
DPL (descriptor privilege level), 47
DPMI (DOS Protected Mode Interface) specification, 24, 74–75, 433
drag and drop operations, 247
driver registration packet (DRP), 298, 433
drivers. *See* device drivers; mini-drivers
drop-down list boxes, 93
DRP (driver registration packet), 298, 433
DuplicateHandle() API function, 240, 241
dynaload (DL) VxDs, 133, 433
dynamic data exchange (DDE)
  defined, 433
  vs. OLE technology, 245
  Win32 APIs, 228
dynamic linking, 71, 82–84
dynamic link libraries (DLLs), 82, 147, 433

E

Eastman Kodak, 260
EGA (Enhanced Graphics Adapter), support for, 67, 433
EISA (Extended Industry Standard Architecture) bus, 310, 311, 434
electronic mail, 23, 28, 349–50, 394–98
embedding, 246, 434. *See also* OLE technology
EndDoc() API function, 270
EndPage() API function, 270
end users, and ease of use, 3, 19–22, 29
enhanced mode, 67, 85

EnterCriticalSection() API function, 239
enumerating, defined, 354, 434
enumerators, and Plug and Play, 324, 328, 334–35
error handling, 249–52. *See also* application program errors
Escape() API function, 269
EtherExpress network adapter, 311
event driven programming, 96–97, 99–100, 434
event logging APIs, Windows 95 vs. Windows NT, 236–37
events
  defined, 76–77, 434
  Plug and Play, 322, 333
  VMM services for, 140
  Win32 APIs, 240
exception handling, 249–52
exceptions, defined, 434
execution priority, 114–15, 116, 117
expanded memory, 38, 74, 85
Explorer program, 169, 196, 434
export table, defined, 434
Extended MAPI, defined, 434
extended memory, 74, 85

F

FAT (File Access Table) filesystem
  defined, 434
  vs. VFAT filesystem, 284–85
  in Windows 95, 275, 282, 284
feature specification, Windows 95, 13–28
file decompression, Win32 APIs, 228
file management subsystem. *See* filesystem
File Manager program, 22, 159, 160–61, 169, 195
file mapping objects, 127–28
filenames, long
  application support, 214, 217, 218
  overview, 23, 213–15, 275, 281–82
  short equivalents, 282, 284–85, 286, 288–89
  storing, 282–88
  Windows 3.1 and MS-DOS applications, 161, 214, 282, 289–91
  Windows 95 vs. Windows NT, 291
File Open dialog box, 210, 211–12
file preview windows, 212
files. *See* folders
file synchronization, 400–404
filesystem
  defined, 66, 434
  layered design, 277–81
filesystem, *continued*
  long filename support, 23, 213–15, 275, 281–91
  and MS-DOS, 112, 276–77, 279
  network support, 275
  new and improved features, 275–77
  subsystem architecture, 277–81
  Windows 3.1 vs. Windows 95, 66, 72, 107
filesystem drivers (FSDs)
  calling, 293–94
  defined, 435
  entry points, 295–96
  network, 352, 440
  overview, 294–96
  VFAT example, 279–80
file viewers, 23
Find and Replace dialog box, 211
FindClose() API function, 290
FindFirstFile() API function, 290
FindNextFile() API function, 290
folders
  defined, 435
  design evolution of, 195–96
  on desktop, 172–73, 177
  overview, 167, 170–71
fonts, system, changes in, 200, 201
Fonts dialog box, 211
foreign languages, 248–49
.FOT files, 258
frame buffer, 267–68, 435
FreeImageColorMatcher() API function, 260
FS_ConnectNetResource() function, 294, 295
FSDs. *See* filesystem drivers
FS_MountVolume() function, 294, 295
function calls, 71

G

gates, 57, 435
Gates, Bill, 18, 19, 191, 198, 211
GDI32.DLL file, 147, 148
GDI (Graphics Device Interface)
  API support, 81, 82
  defined, 65, 435
  device-independent capability, 253–54
  as dynamic link library, 82
  image color matching capability, 254, 259–62
  loading, 134
  metafile support, 258
  overview, 64, 65, 252–55

GDI (Graphics Device Interface), *continued*
privilege level, 108
resource limit expansion, 254, 256–57
subsystem architecture, 255–56
Win32 APIs, 227, 231–32
Windows 95 improvements to, 254–58
GDINFO data structure, 266, 268
GDT (global descriptor table), 41, 42
GDTR register, 41
general protection faults, 53, 56, 117–18
geometry (of a device), defined, 435
GetBrush() API function, 148
GetCurrentDirectory() API function, 290
GetCurrentProcess() API function, 110
GetCurrentTask() API function, 110
GetFileAttributes() API function, 290
GetFileTime() API function, 290
GetLastError() API function, 233
GetMessage() API function, 82, 97
GetStockObject() API function, 148
GetSysColor() API function, 245
GetThreadDesktop() API function, 233
GetVersion() API function, 233
GetVolumeInformation() API function, 290
global context, 70–71, 73
global descriptor table (GDT), 41, 42
global heap, 87
GlobalMemoryStatus() API function, 122
grabber, 264, 446
granularity bit, 47, 435
graphical user interfaces (GUIs), characteristics of, 168–69
graphics coordinate systems. *See* coordinate systems
Graphics Device Interface. *See* GDI
.GRP files, 171, 242
GUIs (graphical user interfaces), characteristics of, 168–69

H

handles, 99, 240, 292, 435. *See also* sizing handle
hardware
device protection, 57–60
dynamic configuration changes, 318
flexibility goal, 320–21
as hardware tree nodes, 329–31
information databases, 332–33
installation and configuration, 20, 315–17
interfacing to, 302–3
hardware, *continued*
platforms, 4–5
and Plug and Play, 315–17, 318, 322, 331–32
hardware tree
building during boot process, 328, 332–33
defined, 322, 436
device nodes, 329–31
Plug and Play example, 325–28
vs. registry, 328, 332
HeapCreate() API function, 129
heaps, 87, 88, 129, 256–57, 436
help system
and application developers, 186, 219
changes in Windows 95, 187–88
context sensitivity of, 188
task-oriented approach of, 187–88
visibility of, 187
Hewlett-Packard LaserJet printers, 273
hidden VM, 73, 112
home base, taskbar as, 179, 180
"hot mouse," 203
hourglass cursor, 27, 188

I

IBM MicroChannel bus, 310, 311, 439
IBM OS/2, 29, 37
IBM PC AT computers, 4–5, 36
IBM Personal Computer, 35, 39, 40
ICM (image color matching), 254, 259–62, 436
icons, 204–5
IEEE (Institute of Electrical and Electronics Engineers), 313, 320
IFS (installable filesystem), 107, 436
IFSMgr_RegisterMount() service, 294
IFSMgr_RegisterNet() service, 294
image color matching (ICM), 254, 259–62, 436
import libraries, 83, 436
independent software vendors (ISVs), 29
.INF files, 322, 332
Info Center, Windows 95, 394–98
.INI files, 242
InitializeCriticalSection() API function, 239
in-place activation, 246, 436
input desynchronization, 119
installable filesystem (IFS), 107, 436
installable filesystem (IFS) manager, 277, 279, 291–94, 352, 436
Institute of Electrical and Electronics Engineers (IEEE), 313, 320

Intel Corporation. *See also names of processors*
EtherExpress network adapter, 311
and Plug and Play standard, 4, 313
processors, 33, 35–37, 38–54
interface. *See* user interface
InterlockedDecrement() API function, 239
InterlockedExchange() API function, 239
InterlockedIncrement() API function, 239
international support, 248–49
interrupt requests (IRQs), 57, 437
interrupts, 57, 141, 290, 301, 436. *See also* software interrupts
interrupt service routines (ISRs), defined, 437
I/O operations, controlling, 57–60
I/O packet (IOP), 297, 299, 437
I/O permission bitmaps, 45, 59, 60
I/O ports, defined, 437
IOREQ data structure, 293
IOS. *See* I/O subsystem
IOS_Register() service, 297, 298, 302
IOS_Requestor_Service service, 297, 300
IOS_SendCommand() service, 297
I/O subsystem (IOS)
defined, 280, 437
device driver initialization, 298–99
port driver example, 302–3
service requests, 297, 300–301
VxD services, 297
I/O trapping VMM services, 141
IPX/SPX protocol, 367, 369, 371, 437
IRQs (interrupt requests), 57, 437
ISA (Industry Standard Architecture), 313, 317, 437
ISRs (interrupt service routines), defined, 437
ISVs (independent software vendors), 29

J, K

Jaguar project, 104
journal records, 270
Kernel
API support, 81, 82
defined, 65, 437
as dynamic link library, 82
loading, 134
privilege level, 108
Win32 APIs, 227
as Windows 95 component, 64, 65
kernel, defined, 437
KERNEL32.DLL file, 147, 148
key depressions, as events, 97

L

languages, foreign, 248–49
LAN Manager, 342, 343, 345, 368
LANs (local area networks), 8–10, 27, 28
LANtastic, 343, 344
laptop computers
docking station support, 399–400
and PCMCIA bus, 312, 313, 318, 320
power management, 398–99
Windows 95 support, 381, 398–400
layered filesystem design, 277–81
LDT (local descriptor table), 41, 42
LDTR register, 41
least recently used (LRU) technique, 122, 437
LeaveCriticalSection() API function, 239
legacy, defined, 437
light source, 184, 199
linear addressing, 25, 39, 85, 143, 438
LineTo() function, 83
linker, 82–83
links, 246, 438
list view control, 210
.LNK files, 171
load group mask (LGM), 299
LoadImageColorMatcher() API function, 260
local area networks (LANs), 8–10, 27, 28
local buses, 312, 320
local descriptor table (LDT), 41, 42
locale, defined, 438
locale APIs, 249
local heap, 87
locality of reference, 50, 438
logical color space, 259, 261–62
logical frame buffer, 267–68
long filenames
application support, 214, 217, 218
overview, 23, 213–15, 275, 281–82
short name equivalents, 282, 284–85, 286, 288–89
storing, 282–88
Windows 3.1 and MS-DOS applications, 161, 214, 282, 289–91
Windows 95 vs. Windows NT, 291
look and feel, 164, 167–69, 438
LRU. *See* least recently used (LRU) technique

M

MAC (media access control) driver, 353, 368, 438
mainframes, 8

MAPI (message application programming interface), 28, 396–97, 438
mapped files, 127–28, 438
MapViewOfFile() API function, 127, 128, 241
MapViewOfFileEx() API function, 127, 128
marketing of Windows 95, 28–30
maximize/restore button, 182, 183, 204
MDI (multiple document interface), 190, 438
media access control (MAC) drivers, 353, 368, 438
memory
  640K barrier, 39–40
  and 80286 protected mode, 41–43
  addressing, 24–25, 45–54
  descriptor format, 45–48
  local vs. global, 87
  protection, 45, 55–56
  segmented architecture, 38–39, 41–43
  segmented vs. linear, 25, 38–39, 85, 143
  virtual vs. physical addresses, 69
memory management
  application, 87–88
  overview, 85–90
  and programming, 241
  system, 88–90
  Virtual Memory Manager (VMM), 125–29
  VMM services, 140
  Win32 APIs, 232, 241
memory mapped devices, defined, 439
memory mapped files, 88, 127–28, 258
memory maps
  original IBM PC, 39, 40
  Win32 applications, 125, 126
  Windows 95, 108–10
menus
  changes in, 202–4
  continuation, 174, 175
  popup, 185, 202–3, 443
  Start menu, 169, 174, 175, 448
message application programming interface (MAPI), 28, 396–97, 438
MessageBox() API function, 83
MessageBoxEx() API function, 249
message loops, defined, 439
message queues, 97, 119–21
messages, 71, 94, 97, 439
message stores, defined, 439
messaging, 396–97, 398, 439
metafiles, 272, 439
MicroChannel bus, 310, 311, 439
Microsoft Corporation. *See also* Gates, Bill
  developer relations groups (DRG), 29
  and development of Plug and Play standard, 4, 312–14
  family of Windows operating systems, 12–13
  importance of OLE technology to, 166, 245–46
  "Ready To Run" campaign, 311
  Windows 95 shell design story, 189–98
  Windows networking history, 342–46
Microsoft LAN Manager, 342, 343, 345, 368
Microsoft Mail, 394, 396
Microsoft OS/2, 2, 4, 6, 7, 36, 37, 44, 82. *See also* IBM OS/2
Microsoft Pen Windows, 23
Microsoft Windows/386, 4, 37, 111
minicomputers, 8
mini-drivers
  defined, 439
  for display driver, 254, 264, 265, 266, 268
  performance of, 92
  for printer driver, 253–54, 272–73
  and VxDs, 91–92
  in Windows 95, 91–92
  in Windows NT, 91
minimize button, 182, 183, 204
miniport drivers, 281, 439
MIPS 4000 processor, 33
modems, 391, 392
MODEMS.INF file, 391
modes, and Windows 95, 67. *See also* protected mode; real mode
modules, defined, 80–81
monitor, 271–72, 375, 377, 439
Motorola processors, 39, 44
mouse
  clicks as events, 97
  double-clicking, 174, 190
  "hot mouse," 203
  right button, 185, 202
MoveFile() API function, 290
MPR (multiple provider router), 351–52, 355–62, 439
MS-DOS–based applications
  and 32-bit addressing, 74
  80386 support for, 44, 59, 60–61
  benefits of virtual mode, 60–61
  and BIOS, 5

MS-DOS–based applications, *continued*
calls to system services, 136–37
compatibility issues, 4, 7, 44, 45
decline of, 2
and device drivers, 58–59, 90
and DPMI standard, 74–75
future of, 23
and I/O permission bitmap, 45, 59, 60
long filenames support, 214, 282, 289–91
and multitasking, 78
in protected mode, 73–75
and real mode drivers, 67, 69, 281, 307–8, 445
running in single application mode, 15, 64, 111–12
running under Windows NT, 33
starting from Windows 95, 59, 60
as VMs, 59–60, 68, 69, 72–73
and Win32 API, 233
Windows 95 support for, 4, 7, 23, 215–16
MS-DOS operating system
and 640K memory limit, 39, 40
as fallback, 15, 64, 111–12
and filesystem, 276–77, 279
future of, 7, 104
and IBM Personal Computer, 35
INT-based software services, 112, 276
limitations of, 6
relationship to Windows 95, 4, 7, 59, 60, 63–64, 111–12
running in single application mode, 15, 64, 111–12
MS-DOS Shell, 159
MS-DOS virtual machines (VMs)
in 32-bit protected mode, 69
context for, 70–71, 72, 73
defined, 106, 111
and DOS extenders, 69
hidden VM, 73
overview, 68, 69, 72–73
replicating PCs running MS-DOS, 72
as single processes, 80, 113
vs. System VM, 68, 69, 80
virtual address space, 109, 110
as Windows 95 component, 64, 66
MsgWaitForMultipleObjects() API function, 240
MS-Net, 342
MSSHRUI DLL, 373
multimedia, Win32 APIs, 228
multiple document interface (MDI), 190, 438
multiple provider router (MPR), 351–52, 355–62, 439
multitasking
cooperative vs. preemptive, 77–78
critical section management, 79–80
defined, 76, 440
managing, with the scheduler, 76–78
and MS-DOS–based applications, 78
network connectivity example, 76
overview, 76–78
print spooling example, 76
and programming, 238–41
use of term, 75
Win32 APIs, 239–41
multithreaded processing, 27, 116, 188–89
mutex (mutual exclusion), 79, 80, 151–52, 240
mutual exclusion (mutex), 79, 80, 151–52, 240
"My Computer," 176

N

named pipe protocol, defined, 440
native mode, 37, 440
NCP protocol, 347
NDIS (Network Driver Interface Specification)
compatibility issues, 369–70
configuration example, 370–72
defined, 352–53, 440
overview, 368–69
nested execution VMM services, 140
NetBEUI transport, 367, 369, 370, 440
NetBIOS protocol, 367, 440
NETRESOURCE data structure, 357–58
NetWare, 29, 342, 343
Microsoft NetWare client for Windows 95, 27, 28, 347
NetWare Lite, 345
network adapter driver VxD, 353
network connections
defined, 354, 430
multitasking example, 76
PC vs. phone-centric, 390
persistent, 354–55, 442
as Windows 95 benefit, 30, 341, 342, 346–47
Network Driver Interface Specification (NDIS). *See* NDIS
network filesystem drivers (FSDs), 352, 440
networking
client-server, 8–10, 28, 341–42, 343–44, 429
configuring, 370–72

networking, *continued*
LANs, 8–10, 27, 28
peer-to-peer, 341, 342, 343, 344–46
printing, 272, 375–77
security issues, 377–79
subsystem architecture, 347–55, 373, 374
terminology, 353–55
Win32 APIs, 228
Windows history, 342–46
and Windows NT, 345
"Network Neighborhood," 176
network providers (NPs)
authentication SPI, 366
defined, 352, 440
device redirection SPI, 364
enumeration SPI, 365–66
interfacing to, 361–62
services of, 363–66
shell SPI, 365
network servers
access control, 373, 378
architecture, 373, 374
defined, 8, 447
minimum configuration, 10
operating systems for, 5, 10, 372
overview, 8–10
for peer-to-peer networking, 372–74
print spooler, 373
requirements for, 8–9
security issues, 8, 373, 377–79
VSERVER software, 373
and Windows NT, 5, 10, 372
network subsystem, Windows 95, 66, 347–55
network transports, 352, 366–72, 441
not-present interrupt, defined, 441
Novell, 341–42, 343
NetWare Lite, 345
protocols, 66, 347, 367
Windows 95 network support, 27, 28, 347
NPCancelConnection() SPI, 364
NPChangePassword() SPI, 366
NPClosedEnum() SPI, 366
NPDeviceMode() SPI, 364
NPDirectoryNotify() SPI, 365
NPDisplayCallback() SPI, 365
NPEndSession() SPI, 364
NPEnumResource() SPI, 366
NPFormatNetworkName() SPI, 365
NPGetCaps() SPI, 364
NPGetDirectoryType() SPI, 365
NPGetDisplayLayout() SPI, 365
NPGetEnumText() SPI, 365
NPGetHomeDirectory() SPI, 366
NPGetNetworkFileProperties() SPI, 365
NPGetResourceParent() SPI, 365, 366
NPGetUser() SPI, 364
NPLogoff() SPI, 366
NPLogon() SPI, 366
NPNotifyAddConnection() SPI, 364
NPOpenEnum() SPI, 366
NPs. *See* network providers
NPSearchDialog() SPI, 365
NP() SPI, 366
NPValidLocalDevice() SPI, 364

O

object filesystem, 247–48, 441
object linking and embedding. *See* OLE technology
object orientation, 11, 100, 166, 247
objects. *See also* control objects; OLE technology
configuring, in property sheets, 185–86
defined, 441
on desktop, 177
property sheets for, 185–86
referencing vs. copying, 171–72
on taskbar, 179
ODBC (Open Database Connectivity), defined, 441
offset address, 46
OLE automation, defined, 441
OLE clients, 167, 188, 246, 247
OLE servers, 246, 247
OLE technology
and application developers, 100, 217–18, 220, 245–48
and Cairo, 220, 246, 247–48
client vs. server applications, 246, 247
compound documents, 167, 212, 246, 247, 430
vs. DDE, 245
defined, 441
and document-centric interface, 159, 166–67, 433
and drag and drop operations, 247
importance of, 166, 245–46
in-place activation, 246
marketing of, 29

OLE technology, *continued*
overview, 22, 245–48
Win32 APIs, 227
online help
and application developers, 186, 219
changes in Windows 95, 187–88
context sensitivity of, 188
task-oriented approach of, 187–88
visibility of, 187
Open Database Connectivity (ODBC), defined, 441
Open Datalink Interface (ODI) specification, 369, 441
Open dialog box, 210, 211–12
OpenFile() API function, 290
OpenFileMapping() API function, 127
OpenGL 3-D graphics library, 228
OpenMutex() API function, 240
OpenSemaphore() API function, 240
operating systems. *See also* base system, Windows 95; MS-DOS operating system
choices in, 6–7
limitations of MS-DOS, 6
and processors, 33, 44–45
protected mode, 23–24
protection capabilities, 54–60
Windows 95 as, 7, 63–64, 66–67
Windows family of, 12–13
OS/2. *See* IBM OS/2; Microsoft OS/2
OS/2 LAN Manager, 368

P

page descriptors (PDs), 123
paged virtual memory, 45, 50–52
page granularity, 47
pages, 121, 123–24, 442
Page Setup dialog box, 213
page tables, 50, 51, 52–53
paging, 45, 50, 122, 436. *See also* demand paging, defined
paragraphs, memory, defined, 38, 442
parent windows, 95
pass through authentication, defined, 442
paths, 257, 442
PC-centric connections, 390
PCI bus, 313, 442
PCL language, 273
PCMCIA bus, 312, 313, 318, 320, 442
PC Network (IBM), 343
PCs, architecture of, 4–5
PDAs (personal digital assistants), 381
PDEVICE data structure, 266
peer-to-peer networking, 27, 66, 341, 342, 343, 344–46
defined, 442
server machines for, 28, 372–74
pen-based applications, 23, 28, 228, 425
Pentium processor
as 386 processor, 37, 44
and backward software compatibility, 36
and virtual 8086 mode, 37
and Windows, 37
performance requirement, Windows 95, 16–17
persistent connections, 354–55, 442
personal digital assistants (PDAs), 381
Phoenix Technologies, and Plug and Play standard, 4, 313
phone-centric connections, 390
physical frame buffer, 267–68
physical memory
and 80286 processor, 41, 42, 43
calculating addresses in protected mode, 45
defined, 442
managing, 88–90, 121–25
vs. virtual memory, 69
pixels, defined, 443
platforms
for 32-bit programs, 5–6
MS-DOS vs. Windows vs. UNIX vs. OS/2, 2
for running Windows, 4–5
Plug and Play BIOS, 315, 317, 324, 336–37, 428
Plug and Play standard
and BIOS, 315, 317, 324, 336–37, 428
and bus design, 315–17
for bus types, 313
compatibility issues, 319–20
defined, 20, 443
goals for, 314–21
history of, 312–14
overview, 4, 309–10
and resource types, 325
why needed, 310–12
Plug and Play subsystem
and application developers, 241–42
components overview, 321–25
device drivers, 324, 337–38
and docking stations, 398–400
hardware tree, 325–32

Plug and Play subsystem, *continued*
printer support, 272
and system setup, 173
use of registry, 243
Win32 APIs, 241–42
point to point protocol (PPP), 385, 443
popup menus, 185, 202–3, 443
portability, of Windows NT, 10
portable computers
docking station support, 399–400
and PCMCIA bus, 312, 313, 318, 320
power management, 399
Windows 95 support, 381, 398–400
port drivers (PDs)
communications, 392–94
defined, 281, 385, 443
execution, 303
initializing, 302–3
and interrupts, 303
overview, 277, 302
porting
16-bit code to Win32, 229–33
tools for, 229–30, 231
PORTTOOL.EXE file, 229
POSIX, defined, 443
PostScript printing, 273
PowerPC processor, 33
PPP (point to point protocol), 385, 443
preemptive multitasking
critical section management, 79–80
defined, 77, 443
problem of 16-bit code, 149–55
scheduling, 77, 78, 100
and Win32 applications, 26–27
present bit, 47, 53
preview windows, 212
primary scheduler, 114
primitives, system, 67, 89–90, 239–40
Print dialog box, 211
printer APIs, Windows 95 vs. Windows NT, 236
printer drivers
DIB engine/mini-driver combination, 253–54, 272–73
dynamic links, 83–84
universal, 24, 253–54, 272–73
printers, configuring, 22, 24
Printer Setup dialog box, 213
printing
API functions for, 236, 269
and bi-directional communication, 272
printing, *continued*
on networks, 375–77
process of, 270–72
subsystem architecture, 269, 270, 272
using shortcut concept, 171
Windows 95 improvements, 269
Print Manager program, 22, 162, 169
print processor, 271
print provider (PP), 375
print request router (PRR), 375, 443
Print Setup dialog box, 211
print spooler, 76, 270, 271, 373, 448
priorities. *See* execution priority
private heaps, 88, 129
privilege levels. *See also* protection rings
for applications, 56
descriptor table entry, 47
for operating system protection, 56–57
switching between, 57, 84
processes
critical sections in, 79–80
defined, 443
MS-DOS VMs as, 80
in System VM, 80
vs. tasks, 76, 110–11
vs. threads, 80, 113
Windows applications as, 80
processor fault VMM services, 141
Program Manager program, 22, 159, 160–62, 169, 177
programming. *See also* application developers
event driven, 95, 96–97, 99–100, 434
and message handling, 97–99
object-oriented, 11, 100, 166, 247
and OLE, 100, 217–18, 220, 245–48
under Windows 95, 99–100
Windows basics, 96–100
progress indicator control, 208
properties, defined, 443–44
Properties menu item, 185
property sheets, 185–86, 209–10, 444
protected mode
and 80286 processor, 36, 41–43
defined, 444
descriptors in, 41, 42
device drivers, 67, 281
and indirect access to memory, 42–43
mapper, 281
MS-DOS–based applications in, 73–75, 281
and operating systems, 23–24

protected mode, *continued*
selectors in, 41, 42
virtual mode as part of, 60
VMM services, 140
protection capabilities
of 80386 processor, 54–60
device protection, 57–60
memory protection, 55–56
operating system protection, 56–57
protection rings
and base system, 84
defined, 444
overview, 107–8
ring zero, 56, 135–40
protocols, defined, 444
protocol stack, defined, 444
PRR (print request router), 375, 443
PulseEvent() API function, 240

Q, R

Quarterdeck, 37
radio buttons, 93
RaiseException() API function, 252
RAM allocation, 127. *See also* physical memory
RAM (random access memory), and virtual memory management, 49–53
RAS (remote access services), defined, 445
RasDial() API function, 389
RasEnumConnections() API function, 389
RasGetConnectStatus() API function, 389
RasHangup() API function, 389
rasterizer, 256, 258, 444
raw input queue, 119, 120, 445
read/write bit, 53
"Ready To Run" campaign, 311
real mode, 41, 45, 60, 67, 112, 445
real mode drivers, 67, 69, 281, 307–8, 445
ReconcileObject API, 403
Recycle Bin feature, 198
redirector, defined, 445
reentrancy, and 16-bit vs. 32-bit code, 149–55
registers, segment. *See* segmented addressing
registration database, 242
registry
application use, 218–19
defined, 445
organization of, 243–44, 332
in Plug and Play subsystem, 322
and programming, 242–44
Win32 APIs, 21, 244
ReleaseMutex() API function, 240
ReleaseSemaphore() API function, 240
remote access services (RAS), defined, 445
remote communications, 28, 382–94
elements of, 383–85
types of access, 386–89
remote network access (RNA), 383, 385–89, 445
remote procedure calls (RPCs), 228, 367, 445
RemoveDirectory() API function, 290
reserving virtual address space, 128–29
ResetEvent() API function, 240
resource arbitrators, 324, 335–36, 445
resources
availability, 55, 87, 237, 256–57
defined, 445
network, defined, 353
usage count, 81
resource sharing, 81, 84
rich text, 209, 446
right mouse button, 185, 202
ring zero, 56, 135–40. *See also* privilege levels; protection rings
RISC processor, defined, 446
RNA (remote network access), 383, 385–89, 445
robustness requirement, 8, 17–18
RPCs (remote procedure calls), 228, 367, 445
runtime memory requirements, Windows 3.1 vs. Windows 95, 87

S

safe driver, defined, 446
scalability, 164, 201–2, 227
ScheduleJob() API function, 236
schedulers
controlling, 116–17
and cooperative multitasking, 77–78
defined, 76, 446
and events, 76
importance of threads, 80, 113
and preemptive multitasking, 78
primary vs. timeslice, 114–16
and priorities, 77, 114–15, 116, 117
and time slices, 76, 77
and Virtual Machine Manager (VMM), 112–21, 141
and VM control flags, 115–16
screen display
3-D appearance, 184, 198–201
controls, 205–10
default, 157, 158, 192–93

screen display, *continued*
design evolution, 189–98
dialog boxes, 210–13
elements of, 201–13
icons, 204–5
menus, 202–4
overall Windows 95 appearance, 22, 182–84, 198–201
scalability, 164, 201–2
scroll boxes, 204, 205
sizing handle, 204, 205
window buttons, 204
screen grabber, 264, 446
scroll bars, 94, 205
scroll boxes, 204, 205
SCSI_CONFIGURATION_INFORMATION data structure, 306
SCSI device support, 201, 304, 306–7
SCSI_INITIALIZATION_DATA data structure, 306
SCSIizer, 281
SCSI manager, 281, 306–7, 446
SCSI_REQUEST_BLOCK data structure, 306
SCSI (Small Computer System Interface) bus, 312, 313, 320, 446
SDI (single document interface), 190, 446
security APIs, Windows 95 vs. Windows NT, 234
security of servers, 8, 373, 377–79
Security Provider, 373
segment bit, 47
segmented addressing
for 8086 processor, 37, 38
for 80286 processor, 41–43
architecture, 38–39, 41–43
defined, 447
vs. linear addressing, 25, 38–39, 85, 143
segment registers. *See* segmented addressing
segments, 38–39, 41–43, 45, 46, 47–48, 447
selectors, 41, 42, 46, 87, 109
semaphores
defined, 240, 447
Win16mutex, 152–55
and Win16 subsystem, 151–52
Win32 APIs for, 240
serial ports, adding, 57
server APIs, Windows 95 vs. Windows NT, 236
server applications, defined, 447
servers. *See* network servers
service control manager APIs, Windows 95 vs. Windows NT, 236
service provider interface (SPI), 348, 447
service providers, 348, 447
service tables, VxD, 130, 447
SetCurrentDirectory() API function, 290
SetEvent() API function, 240
SetFileAttributes() API function, 290
SetFileTime() API function, 290
shading, 184, 199–200
shared memory, 86, 88, 127–28, 447
share-level security, 378, 379, 447
share name, defined, 448
share points, 354, 373, 448
shell
3-D appearance of, 184, 198–200
as 32-bit application, 147, 188
animation in, 196–97
briefcase object in, 400–404, 428
defined, 159, 448
design retrospective, 189–98
development of, 190–92
elements of, 169–82
extensibility of, 189
need for application consistency with, 219
new features, 22–23
for novice vs. experienced users, 192, 193
as OLE client, 167, 188
outside influences on, 189–90
prototyping in Visual Basic, 190–92
system color scheme, 200
system fonts, 200, 201
threading capabilities of, 188–89
and transfer model, 191, 197–98
usability testing of, 189–90
Win32 APIs, 228
as Windows 95 component, 64, 65
_SHELL_BroadcastSystemMessage service, 135
_SHELL_CallAppyTime service, 134
_SHELL_HookSystemBroadcast service, 135
_SHELL_PostMessage service, 135
Shell VxD, 134–35, 448
shortcuts, 170, 171–72, 448
Simple MAPI, defined, 448
single MS-DOS–based application mode, 15, 64, 112
sizing handle, 204, 205
sizing windows, 205
slider control, 208
SMB protocol, 66, 448
sockets, 228, 367, 371, 448
software. *See* applications

software interrupts, 72, 73, 136, 137
specification for Windows 95, 13–28
spin boxes, 93, 208–9
SPI (service provider interface), 348, 447
spooler, 76, 270, 271, 373, 448
standard mode, 67
Start button, 174
StartDoc() API function, 270
Start menu, 169, 174, 175, 448
StartPage() API function, 270
startup screen. *See also* screen display
  design evolution, 192–93
  Windows 95 vs. Windows 3.1, 157, 158
static VxD, defined, 448
status window control, 206–7
streams, 247
structured exception handling, 249–52, 448
swap faults, 50
swap file, 50, 123, 124, 449
synchonization VMM services, 141
synchronization primitives, 239–40
system bus design, 310
system crashes, 17–18
system file handle structure, 292
system fonts, changes in, 200, 201
SYSTEM.INI file, 21, 242, 243, 268, 298, 361, 362
system menu. *See* window menu
System Policy Editor utility, 378
system primitives, 67, 89–90, 239–40
system reentrancy, and 16-bit vs. 32-bit code, 149–55
system resources. *See* resources
system taskbar, 174, 179–81, 194–95
system tray, defined, 449
System Virtual Machine (VM)
  context for, 71
  defined, 65, 106, 111, 449
  vs. MS-DOS VMs, 68, 69, 80
  multiple processes in, 80
  overview, 71–72
  scheduling within, 116
  and Win32 applications, 71–72
  as Windows 95 component, 64, 65, 68, 69

T

tab control, 209
TAPI (Windows Telephony API), 382–83, 385, 389–90, 449
taskbar
  as anchor point, 179, 180
  and application compatibility, 180–81
taskbar, *continued*
  buttons on, 194, 195
  configuring, 180
  default for, 179
  defined, 174, 449
  design evolution, 194–95
  hiding/displaying, 179–80
  overview, 179–81
Task Database (TDB), 80
Task Manager, 161–62, 169
tasks
  defined, 76, 449
  as gates, 57
  vs. processes, 76, 110–11
  use of term, 75
TCP/IP protocol, 66, 347, 371, 449
telephony applications. *See* TAPI (Windows Telephony API)
third-party vendors, 189
threads
  and application errors, 117–18
  and background activities, 118
  as basic unit of scheduling, 80, 113, 114–15
  defined, 449
  execution priority, 114–15, 116, 117
  and general protection faults, 117–18
  limits of, 113
  MS-DOS VMs as, 113
  multiple, 27, 116, 188–89
  overview, 113
  vs. processes, 80, 113
  suspending, 114
  synchronization primitives, 239–40
  in System VM, 116
  and UAEs, 117–18
thumbnails, defined, 449
thunk compiler, 145, 146–47
thunks
  defined, 54, 144, 449
  origin of term, 144
  in Windows 3.1, 144–45
  in Windows 95, 145–47, 148, 149
tiling, 143
timeslices, 76, 77, 449
timeslice scheduler, 114–15, 116
toolbar, 94, 95, 204, 205–6
TOPS networking program, 343
transfer model, 191, 197–98, 449
transports. *See* network transports
traps, as gates, 57
tree view control, 210

TrueType rasterizer, 256, 258, 444
TSD (type specific driver), 280, 304, 305–6, 450
TSR programs, 7
.TTF files, 258
type specific driver (TSD), 280, 304, 305–6, 450

U

UAEs (Unrecoverable Application Errors), 2, 17, 56, 117–18, 450
UNC (Universal Naming Convention), 218, 450
Unicode
  vs. ANSI character set, 235
  defined, 450
  and internationalization, 248
  Windows NT vs. Windows 95 APIs, 235
Unimodem, 385, 391, 450
UNIMODEM.386 driver, 391
universal client, defined, 346–47
universal drivers, defined, 450
Universal Naming Convention (UNC), 218, 450
universal printer driver, 24, 253–54, 272–73
UNIX, 2, 6, 37, 44, 82, 347, 450
Unrecoverable Application Error message, 2, 17, 56
usability tests, for Windows 95 shell, 189–90
User
  API support, 81, 82
  defined, 66, 450
  as dynamic link library, 82
  loading, 134
  privilege level, 108
  Win32 APIs, 227
  as Windows 95 component, 64, 66
USER32.DLL file, 147, 148
user accounts, defined, 450
user interface. *See also* shell
  important characteristics, 168–69
  improvements to, 159–65
  look and feel issue, 167–69
  for Plug and Play subsystem, 324
  Win32 APIs, 245
  Windows 95 design, 95–96
  Windows overview, 92–95
user-level security, 378, 379, 450
users, active, defined, 30
user/supervisor bit, 53
utility functions, 24

V

VCACHE, defined, 451
VCOMM VxD, 385, 391, 392–94, 451
VCPI (Virtual Control Programming Interface) specification, 74
vendors. *See* independent software vendors (ISVs)
vendor supplied driver (VSD), 280–81, 451
VFAT filesystem
  defined, 451
  driver, 279–80
  vs. FAT filesystem, 284–85
VFLATD VxD, 267–68, 451
VGA, defined, 451
video display, controlling with DIB engine, 262
video memory, 264, 267–68
View menu, 203, 204
virtual 8086 machine. *See* virtual 8086 mode
virtual 8086 mode. *See also* virtual mode
  and 80386 processor, 37, 44, 45, 68
  defined, 451
  and virtual machines (VMs), 59, 68
  vs. virtual memory, 48
  vs. Windows VM, 68
virtual addresses
  defined, 451
  vs. physical addresses, 69
virtual address space
  for 16-bit Windows applications, 109
  for 32-bit Windows applications, 25, 27, 85, 86–88, 109, 110, 125, 126
  defined, 451
  for MS-DOS VM, 109, 110
  reserving, 128–29
  shared vs. private, 86
  system memory map, 108–10
  system vs. application, 86
VirtualAlloc() API function, 128–29
Virtual Control Programming Interface (VCPI) specification, 74
virtual device drivers (VDDs), 91–92, 108
virtual device drivers (VxDs), 67, 84–85
  callback mechanism, 131–32
  calling between, 136, 138–40
  defined, 67, 452
  defining service tables, 130
  dynaload, 133, 433
  loading dynamically, 133–34
  loading in Windows 3.1, 132–33
  loading in Windows 95, 133–34
  mini-drivers as, 91–92
  and protection rings, 108
  Shell VxD, 134–35, 448
  VMM services to, 117, 130, 133, 140–41

Virtual Machine Manager (VMM)
callback mechanism, 131–32, 144
calling of services, 84, 131–35, 136
configuration manager services, 141
debug services, 141
defined, 67, 106, 451
event services, 140
I/O trapping services, 141
memory management services, 140
nested execution services, 140
new features in Windows 95, 111, 125–29
overview, 111, 125
processor fault services, 141
processor interrupt services, 141
protected mode execution services, 140
registry services, 140
and scheduling, 112–21, 141
scope of services, 130
services of, 129–41
services to VxDs, 117, 130, 133, 140–41
synchonization services, 141
system mapping function, 109
VM callback services, 141
VM interrupt services, 141
VMs as clients, 131
as VxD, 129
virtual machines (VMs). *See also* Windows virtual machines
context for, 68, 70–71, 72, 73
control blocks, 130
control flags, 115–16
defined, 451
MS-DOS–based applications as, 59–60, 68, 69, 72–73
overview, 68–75
virtual memory
defined, 48, 451
managing, 49–53
paged, 45, 50–52
vs. virtual mode, 48
and Win32 applications, 86–88
virtual mode
benefits for MS-DOS–based applications, 60–61
defined, 48
as part of protected mode, 60
vs. virtual memory, 48
Visual Basic, prototyping Windows 95 shell in, 190–92
visual cues, 183, 451
visual design issues, 164–69
VMM. *See* Virtual Machine Manager
VMMcall macro, 139
VMs. *See* virtual machines (VMs)
VMStat_Background flag, 115
VMStat_Exclusive flag, 115
VMStat_High_Pri_Background flag, 115
voice mail, and Windows 95 Info Center, 394–98
volume request packet (VRP), 299–300
volume tracking driver (VTD), 280, 304–5
VSERVER software, 373
VTD. *See* volume tracking driver
VxDcall macro, 139
VXDLDR module, 134
VxDs. *See* virtual device drivers (VxDs)

W

WaitForMultipleObjects() API function, 240
WaitForMultipleObjectsEx() API function, 240
WaitForSingleObject() API function, 240
WaitForSingleObjectEx() API function, 240
Wastebasket feature, 198
widening, defined, 452
Win16Lock, defined, 452
Win16Mutex
defined, 452
drawbacks of, 154–55
as reentrancy safeguard, 152–53
Win16 subsystem, defined, 452. *See also* 16-bit Windows applications
WIN32.386 file, 147
Win32 API
and 16-bit applications, 110, 111, 142
benefits of, 26–27
binaries issue, 225
common dialog functions, 227
communications functions, 228
compatibility issues, 110–11, 224–25
components of, 227–28
controls, 227
DDE functions, 228
and dynamic memory allocation, 87–88
extensibility of, 226
file decompression functions, 228
file location, 147–48
functions unsupported in Windows 95, 234–38
GDI functions, 227, 231–32
goals for, 226–27

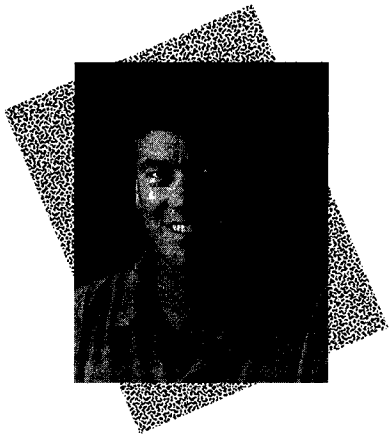
Win32 API, *continued*
graphics APIs, 257
Kernel functions, 227
locale functions, 249
and memory management, 85–90, 232, 241
messaging functions, 396, 397
multimedia functions, 228
multitasking functions, 239–41
named functions, 82
names of functions, 231–32
networking functions, 228, 356–61
and nonportable functions, 233
OLE functions, 227
overview, 25, 65, 81–85, 224–25
pen functions, 228
portability of, 226
porting to, 229–33
preferred Windows API, 99
and registry, 244
RNA Session API, 388–89
RPC functions, 228
scalability of, 227
shell functions, 228
size of, 227
sockets functions, 228
as "standard," 224–25
support for, 226
Telephony API (TAPI), 382–83, 385, 389–90
User functions, 227
and user interface, 245
using, in Windows 95 programming, 238–52
version checking, 233
Win32c subset, 25, 224
Win32s subset, 25, 26, 224, 452
and Windows 3.1, 224
and Windows 95, 26, 81–85, 225, 229–38
Windows NT support, 25
Windows vs. MS-DOS, 82
Win32c API, 25, 224
Win32s API, 25, 26, 224, 452
Win32 Software Development Kit, 144
Win32 subsystem, 144, 147–49, 452. *See also* 32-bit Windows applications
WINBOOT.SYS file, 112
window menu, 203, 452
window procedures, defined, 452
windows
hierarchy of, 95
ownership of, 95
windows, *continued*
parent vs. child, 95
scaling, 164, 201–2
sizing, 205
User as manager for, 66
Windows 3.0, 1, 2, 17
Windows 3.1
calls to system services, 136–37, 139
as cooperative multitasking system, 77–78
files and directories, 167, 171
getting started, 173–74
improvements on, 159–65
installing Windows 95 on existing systems, 20
and Intel 386 chip, 1
physical memory manager, 124
printer control, 24, 162, 163–64
property information, 162
real vs. protected modes, 23
reason for introduction, 2
reliance on MS-DOS, 72
sales of, 2
system management inconsistencies, 160–62
Task Database (TDB), 80
and Win32 API, 224
Windows 95 API compatibility, 65
Windows 95
32-bit application support, 5–6, 24–27, 54
and 80386 processor, 44–45
API coverage, 81–85, 234–38
application guidelines for, 217–20
areas for improvement, 19–28
Beta-1 release, xxv–xxvi, 30
boot process, 112
character set, 235
and client-server systems, 9, 11, 346, 347
codename "Chicago," xxv–xxvi, 1
communications subsystem, 382–94
compatibility requirement, 4, 7, 14–15, 44, 45
as complete operating system, 4, 7, 63–64, 66–67
components of, 106–11
configuration files, 243, 332
coordinate system, 232, 235, 257
and ease of use, 3, 19–22, 29
feature specification for, 13–28
filesystem architecture, 277–81
GDI improvements, 254–58
general goals, 3
getting started, 173–76

Windows 95, *continued*
initial desktop, 174–76
internationalization of, 248–49
marketing, 13, 28–30
Microsoft requirements for, 14–19
minimum hardware requirements for, 9, 16, 67, 245, 413
mission for, 3–6
and MS-DOS compatibility, 4, 7, 44, 45
naming of, xxvi
network architecture, 347–55, 373, 374
and network connectivity, 30, 341, 342, 346–47
networking security, 8, 373, 377–79
new screen display look, 198–213
and object-oriented programming, 100, 111, 166, 247
OLE in, 100, 217–18, 220, 245–48
online help system, 186–88
part of Windows family of operating systems, 12–13
Plug and Play subsystem, 321–38
press rollout, 30
programming under, 99–100
relationship with MS-DOS, 4, 7, 59, 60, 63–64, 111–12
release date, 18–19
resource availability, 55, 87, 237, 256–57
robustness requirement, 17–18
run "as well as Windows 3.1" requirement, 16–17
shell, 169–82
similarity to Cairo, 11
system architecture, 64–66, 104–6
system overview, 63–67
system setup, 173–74
timely availability, 18–19
Windows/386, 4, 37, 111
Windows API functions, 82, 83
Windows-based applications. *See* 16-bit Windows applications; 32-bit Windows applications
Windows device driver, 4, 91
Windows for Workgroups, 27, 28, 66, 107, 112, 197, 342, 343, 345
Windows NT, 4, 5, 6
and 16-bit applications, 33
32-bit support, 25
Advanced Server version, 5, 372
and Cairo, 10
Windows NT, *continued*
defined, 452
dynamic linking capability, 82
graphic coordinates system, 232
Intel processor emulation, 33
minimum hardware requrements for, 9
and networking, 66, 345
Plug and Play support, 313
portability of, 10
as preemptive multitasking system, 77
preexistence of, 104
running MS-DOS applications on, 33
as server machine, 10
variety of processor types for, 33
and Win32 API, 224, 225, 234–38
Windows Open Services Architecture (WOSA), 348–51, 389, 452
Windows Sockets, 228, 367, 371, 453
Windows subsystem. *See* GDI (Graphics Device Interface); Kernel; User
Windows Telephony API (TAPI), 382–83, 385, 389–90
Windows user interface. *See* user interface
Windows virtual machines. *See also* MS-DOS virtual machines (VMs); System Virtual Machine (VM)
address space, 69
defined, 68
importance of, 68–69
initialization, 70–71
vs. Intel virtual 8086 machines, 68
MS-DOS VMs, 68, 69, 72–73
overview, 70–73
System VM, 68, 69, 71–72
Windows VMs. *See* Windows virtual machines
WINDOWSX.H header file, 230
WIN.INI file, 21, 242, 243
WinNet functions, 356–61
WinSock interface, 228, 367
WM_DEVICEBROADCAST message, 238, 242
WM_DISPLAYCHANGED message, 245
WM_KBDLAYOUTCHANGE message, 238
WNetAddConnection() API function, 358, 359
WNetAddConnection2() API function, 358, 359
WNetAuthenticationDialog() API function, 360
WNetCachePassword() API function, 360
WNetCancelConnection() API function, 359
WNetCancelConnection2() API function, 359
WNetCloseEnum() API function, 359

WNetConnectionDialog() API function, 359
WNetDeviceGetFreeDevice() API function, 360
WNetDeviceGetNumber() API function, 360
WNetDeviceGetString() API function, 360
WNetDisconnectDialog() API function, 359
WNetEnumResource() API function, 359
WNet functions, 356–61
WNetGetConnection() API function, 359
WNetGetLastError() API function, 360
WNetGetSectionName() API function, 360
WNetNotifyRegister() API function, 359
WNetOpenEnum() API function, 359
WNetSetLastError() API function, 360
WNetUNCGetItem() API function, 360
WNetUNCValidate() API function, 360
working set, defined, 453
WOSA (Windows Open Services Architecture), 348–51, 389, 452
WriteProcessMemory() API function, 123, 241

X–Z

XENIX, 36
yielding, defined, 453
Z order, defined, 453



Adrian King is a native of London and graduated in 1976 from the University of Liverpool with a master's degree in computer science. That same year he joined the European consulting firm Logica, working in its system software division on real time control and communications projects. While at Logica, he founded the Software Products Group, which became Microsoft's European XENIX partner in 1981. Adrian moved to the U.S. in 1984 to become Microsoft's XENIX product manager.

At Microsoft, Adrian worked for Steve Ballmer as XENIX product manager and later became director of operating systems products, assuming responsibilities for MS-DOS and Microsoft OS/2. He later managed the group that developed Windows/386, the product that pioneered the use of software virtual machine technology in Microsoft operating systems.

In the late 1980s Adrian took over product responsibility for the SQL Server and Communications Server products and later Microsoft LAN Manager. In July 1991 he left Microsoft to become vice president of engineering at Artisoft. While he was in charge of development at Artisoft, LANtastic—Artisoft's local area network product—won *PC Magazine*'s Editors Choice award.

In 1992 Adrian founded Gravity Communications, a consulting firm specializing in the preparation of technical literature. He has written the book *Running LANtastic* (Bantam, 1991) and articles for *Microsoft Systems Journal* and other computer magazines.

Adrian is an active general aviation pilot and participates enthusiastically in soccer, skiing, golf, and other sports.

The manuscript for this book was prepared and submitted to Microsoft Press in electronic form. Text files were prepared using Microsoft Word 2.0 for Windows. Pages were composed by Microsoft Press using Aldus PageMaker 5.0 for Windows, with text in New Baskerville and display type in Helvetica Bold. Composed pages were delivered to the printer as electronic prepress files.

Cover Designer
Clement Mok designs, Inc.

Interior Graphic Designer
Kim Eggleston

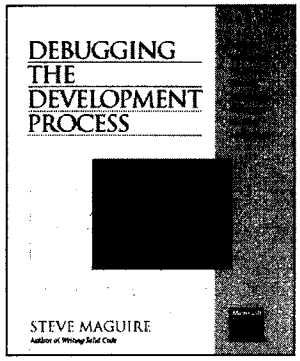
Interior Graphic Artists
David Holter, Sandi Lage, Jim Kramer

Principal Typographer
Barb Runyan

Principal Proofreader/Copy Editor
Shawn Peck

Indexer
Julie Kawabata

Programming Practices



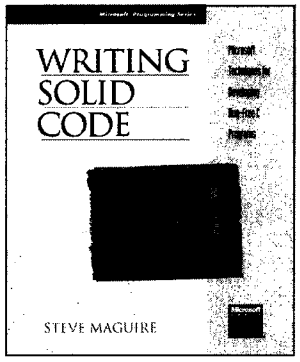
Debugging the Development Process

Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams

Steve Maguire

From the author of the award-winning *Writing Solid Code* comes a compelling look at the people who develop the code and the group dynamics behind the scenes of the software development process. Steve Maguire draws on his real-world experiences at Microsoft for candid accounts of how he brought together and maintained efffective teams for development of timely, high-quality commercial applications. Find out what did and didn't work at Microsoft, and why.

216 pages, softcover $24.95 ($32.95 Canada) ISBN 1-55615-650-2



Writing Solid Code

Microsoft® Techniques for Developing Bug-Free C Programs

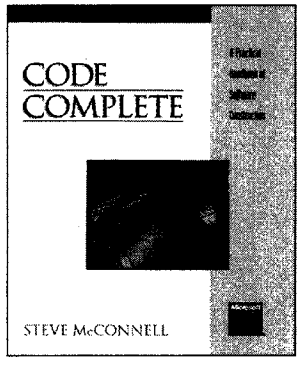
Steve Maguire

Foreword by Dave Moore,
Director of Development, Microsoft Corporation

"I read it with great interest for hours at a stretch. It presents detailed solutions to real problems." **IEEE Micro**

Written by a former Microsoft developer and troubleshooter, this book is an insider's view of the most important aspect of the development process: preventing and detecting bugs. Maguire identifies the places developers typically make mistakes, offers practical advice for detecting costly errors, and presents proven programming techniques for producing clean code.

288 pages, softcover $24.95 ($32.95 Canada) ISBN 1-55615-551-4



Code Complete

Steve McConnell

"We were impressed....A pleasure to read, either straight through or as a reference." **PC Week**

This practical handbook of software construction covers the art and science of the entire development process, from design to testing. Examples are provided in C, Pascal, Basic, FORTRAN, and Ada—but the focus is on programming techniques. Topics include up-front planning, applying good design techniques to construction, using data effectively, reviewing for errors, managing construction activities, and relating personal character to superior software.

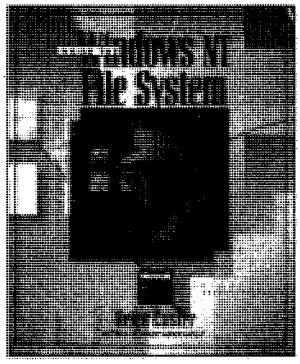
880 pages, softcover $35.00 ($44.95 Canada) ISBN 1-55615-484-4

Microsoft Press®

Microsoft Press® books are available wherever quality books are sold and through CompuServe's Electronic Mall—GO MSP.
Call 1-800-MSPRESS for direct ordering information or for placing credit card orders.*
Please refer to BBK when placing your order. Prices subject to change.

*In Canada, contact Macmillan Canada, Attn: Microsoft Press Dept., 164 Commander Blvd., Agincourt, Ontario, Canada M1S 3C7, or call 1-800-667-1115.
Outside the U.S. and Canada, write to International Sales, Microsoft Press, One Microsoft Way, Redmond, WA 98052-6399.

More Inside Information

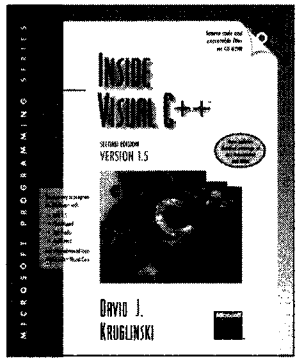


Inside the Windows NT™ File System

Helen Custer

This detailed, informative monograph by critically acclaimed author Helen Custer is an up-to-date adjunct to her bestselling *Inside Windows NT*. In this special edition, Custer expands on her discussion of the robust new Windows NT File System (NTFS) and documents its arduous design and creation process. This book includes the first discussion of data compression in Windows NT, describes the file system's internal structure, and explains in detail how NTFS recovers a volume and reconstructs itself after a system failure.

104 pages, softcover $9.95 ($12.95 Canada) ISBN 1-55615-660-X

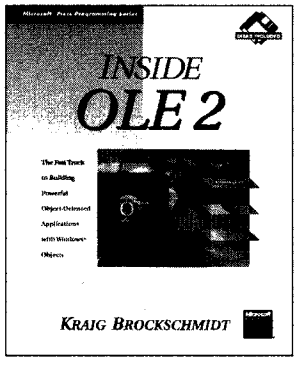


Inside Visual C++,™ 2nd ed.

David J. Kruglinski

Now updated to cover Visual C++ version 1.5, this book discusses OLE, ODBC enhancements, and Microsoft Foundation Class (MFC) Library version 2.5. This is the foundation book for Visual C++ developers programming in Windows. Through lively examples, this book takes readers from the basics through the advanced capabilities of this rich programming environment, while explaining the methodologies and the tools. The CD-ROM includes all the source code files necessary to create the sample programs in the book.

768 pages, softcover with one CD-ROM disk
$39.95 ($53.95 Canada) ISBN 1-55615-661-8



Inside OLE 2

Kraig Brockschmidt

Here's the inside scoop on how to build powerful object-oriented applications for Windows. Written by a leading OLE expert, this guide shows experienced programmers how to take advantage of OLE to develop next-generation applications that will take Windows to a new level. Brockschmidt explains how to build OLE applications from scratch as well as how to convert existing applications. The disks contain 44 source code examples that demonstrate how to implement objects and how to integrate OLE features into your applications.

1008 pages, softcover with two 1.44-MB 3.5-inch disks
$49.95 ($67.95 Canada) ISBN 1-55615-618-9

Microsoft Press®

Microsoft Press® books are available wherever quality books are sold and through CompuServe's Electronic Mall—GO MSP. Call 1-800-MSPRESS for direct ordering information or for placing credit card orders.*

Please refer to BBK when placing your order. Prices subject to change.

*In Canada, contact Macmillan Canada, Attn: Microsoft Press Dept., 164 Commander Blvd., Agincourt, Ontario, Canada M1S 3C7, or call 1-800-667-1115. Outside the U.S. and Canada, write to International Sales, Microsoft Press, One Microsoft Way, Redmond, WA 98052-6399.

APPENDIX C

The Windows Interface Guidelines — A Guide for Designing Software

Microsoft® Windows®

February 1995

This is a preliminary release of the documentation. It may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft Corporation makes no warranties, either expressed or implied, in this prerelease document.

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property rights.

Copyright © 1995 by Microsoft Corporation. All rights reserved.

Microsoft, MS, and MS-DOS, Windows, and the Windows logo are registered trademarks and Windows NT is a trademark of Microsoft Corporation.

Contents

Introduction xiii
What's New xiii
How to Use This Guide xiv
How to Apply the Guidelines xiv
Notational Conventions xv

Chapter 1 Design Principles and Methodology 1
User-Centered Design Principles 1
User in Control 1
Directness 2
Consistency 2
Forgiveness 3
Feedback 4
Aesthetics 4
Simplicity 4
Design Methodology 5
A Balanced Design Team 5
The Design Cycle 5
Usability Assessment in the Design Process 8
Understanding Users 10
Design Tradeoffs 11

Chapter 2 Basic Concepts 13
Data-Centered Design 13
Objects as Metaphor 13
Object Characteristics 14
Relationships 14
Composition 15
Persistence 15
Putting Theory into Practice 15

Chapter 3 The Windows Environment 17
The Desktop 17
The Taskbar 17
The Start Button 18
Window Buttons 18
The Status Area 19

Icons 20
Windows 22

Chapter 4 Input Basics 23
Mouse Input 23
Mouse Pointers 23
Mouse Actions 24
Keyboard Input 26
Text Keys 26
Access Keys 27
Mode Keys 27
Shortcut Keys 28
Pen Input 29
Pen Pointers 32
Pen Gestures 32
Pen Recognition 33
Ink Input 33
Targeting 34

Chapter 5 General Interaction Techniques 35
Navigation 35
Mouse and Pen Navigation 35
Keyboard Navigation 35
Selection 37
Selection Feedback 37
Scope of Selection 38
Hierarchical Selection 38
Mouse Selection 38
Pen Selection 43
Keyboard Selection 44
Selection Shortcuts 45
Common Conventions for Supporting Operations 46
Operations for a Multiple Selection 46
Default Operations and Shortcut Techniques 46
View Operations 47
Editing Operations 49
Editing Text 49
Handles 51
Transactions 51
Properties 53

Pen-Specific Editing Techniques 53

Transfer Operations 59
Command Method 60
Direct Manipulation Method. 63
Transfer Feedback 68
Specialized Transfer Commands. 70
Shortcut Keys for Transfer Operations 70
Scraps 71
Creation Operations 71
Copy Command. 71
New Command 71
Insert Command 71
Using Controls. 72
Using Templates 72
Operations on Linked Objects. 72

Chapter 6 Windows 75
Common Types of Windows. 75
Primary Window Components 75
Window Frames. 76
Title Bars. 76
Title Bar Icons. 77
Title Text. 78
Title Bar Buttons 80
Basic Window Operations 81
Activating and Deactivating Windows 81
Opening and Closing Windows. 82
Moving Windows 83
Resizing Windows. 84
Scrolling Windows. 86
Splitting Windows 92

Chapter 7 Menus, Controls, and Toolbars 97
Menus 97
The Menu Bar and Drop-down Menus 97
Common Drop-down Menus 99
Pop-up Menus 101
Pop-up Menu Interaction 102
Common Pop-up Menus. 103
Cascading Menus. 107
Menu Titles 107

Menu Items 108

Controls 112
Buttons 113
List Boxes 120
Text Fields 127
Other General Controls 132
Pen-Specific Controls 136
Toolbars and Status Bars 139
Interaction with Controls in Toolbars and Status Bars 140
Support for User Options 140
Common Toolbar Buttons 142

Chapter 8 Secondary Windows 145
Characteristics of Secondary Windows 145
Appearance and Behavior 145
Window Placement 148
Modeless vs. Modal 148
Default Buttons 148
Navigation in Secondary Windows 149
Validation of Input 151
Property Sheets and Inspectors 151
Property Sheet Interface 151
Property Sheet Commands 153
Closing a Property Sheet 154
Property Inspectors 155
Properties of a Multiple Selection 156
Properties of a Heterogeneous Selection 156
Properties of Grouped Items 156
Dialog Boxes 157
Dialog Box Commands 157
Layout 157
Common Dialog Box Interfaces 158
Palette Windows 167
Message Boxes 169
Message Box Types 169
Command Buttons in Message Boxes 171
Message Box Text 173
Pop-up Windows 174

Chapter 9 Window Management 175
Single Document Window Interface 175
Multiple Document Interface 176
Opening and Closing MDI Windows 178
Moving and Sizing MDI Windows 178
Switching Between MDI Child Windows 180
MDI Alternatives 181
Workspaces 182
Workbooks 184
Projects 185
Selecting a Window Model 186
Presentation of Object or Task 187
Display Layout 188
Data-Centered Design 188
Combination of Alternatives 188

Chapter 10 Integrating with the System 189
The Registry 189
Registering Application State Information 190
Registering Application Path Information 192
Registering File Extensions 193
Supporting Creation 199
Registering Icons 200
Registering Commands 201
Enabling Printing 202
Registering OLE 202
Registering Shell Extensions 202
Supporting the Quick View Command 204
Registering Sound Events 205
Installation 206
Copying Files 206
Making Your Application Accessible 208
Designing Your Installation Program 208
Uninstalling Your Application 209
Installing Fonts 210
Installing Your Application on a Network 211
Supporting Auto Play 211

System Naming Conventions 213
Taskbar Integration 214
Taskbar Window Buttons 214
Status Notification 214
Message Notification 215
Recycle Bin Integration 216
Control Panel Integration 216
Adding Control Panel Objects 216
Adding to the Passwords Object 217
Plug and Play Support 217
System Settings and Notification. 218
Modeless Interaction 218

Chapter 11 Working with OLE Embedded and OLE Linked Objects 219
The Interaction Model 219
Creating OLE Embedded and OLE Linked Objects 221
Transferring Objects 221
Inserting New Objects 225
Displaying Objects 229
Selecting Objects 232
Accessing Commands for Selected Objects 234
Activating Objects 236
Outside-in Activation 236
Inside-out Activation 237
Container Control of Activation 237
OLE Visual Editing of OLE Embedded Objects. 238
The Active Hatched Border 243
Menu Integration 244
Keyboard Interface Integration 247
Toolbars, Frame Adornments, and Palette Windows 248
Opening OLE Embedded Objects 251
Editing an OLE Linked Object 254
Automatic and Manual Updating 256
Operations and Links 256
Types and Links. 257
Link Management 257
Accessing Properties of OLE objects. 257
The Properties Command 258
The Links Command 260

Converting Types 262
Using Handles 263
Undo Operations for Active and Open Objects 264
Displaying Messages 266
Object Application Messages 266
OLE Linked Object Messages 269
Status Line Messages 271

Chapter 12 User Assistance 273
Contextual User Assistance. 273
Context-Sensitive Help. 273
Guidelines for Writing Context-Sensitive Help 276
Tooltips 277
Status Bar Messages. 277
Guidelines for Writing Status Bar Messages 279
The Help Command Button 279
Task Help 280
Task Topic Windows 280
Guidelines for Writing Task Help Topics 282
Shortcut Buttons 282
Reference Help 283
The Reference Help Window 283
Guidelines for Writing Reference Help 285
The Help Topics Browser 286
The Help Topic Tabs 286
Guidelines for Writing Help Contents Entries 290
Guidelines for Writing Help Index Keywords 290
Wizards 291
Wizard Buttons 291
Guidelines for Writing Text for Wizards 291
Guidelines for Writing Text for Wizard Pages 294

Chapter 13 Visual Design 295
Visual Communication 295
Composition and Organization 295
Color 297
Fonts 299
Dimensionality 300

Design of Visual Elements 300
Basic Border Styles 300
Window Border Style 302
Button Border Styles 302
Field Border Style 303
Status Field Border Style 304
Grouping Border Style 305
Visual States for Controls 305
Layout 312
Font and Size 312
Capitalization 315
Grouping and Spacing 315
Alignment 316
Button Placement 316
Design of Graphic Images 317
Icon Design 318
Pointer Design 320
Selection Appearance 322
Highlighting 322
Handles 323
Transfer Appearance 324
Open Appearance 325
Animation 325

Chapter 14 Special Design Considerations 327
Sound 327
Accessibility 328
Visual Disabilities 329
Hearing Disabilities 330
Physical Movement Disabilities 330
Speech or Language Disabilities 330
Cognitive Disabilities 330
Seizure Disorders 330
Types of Accessibility Aids 331
Compatibility with Screen Review Utilities 332
The User's Point of Focus 334
Timing and Navigational Interfaces 335
Keyboard and Mouse Interface 337
Documentation, Packaging, and Support 337
Usability Testing 338

Internationalization. 338
Text. 339
Graphics 340
Keyboards 341
Character Sets 341
Formats 342
Layout 343
References to Unsupported Features 343
Network Computing 343
Leverage System Support 343
Client-Server Applications 344
Shared Data Files 344
Records Processing 344
Telephony 345
Microsoft Exchange 346
Coexisting with Other Information Services 346
Adding Menu Items and Toolbar Buttons 347
Supporting Connections 347
Installing Information Services 348

Appendix A Mouse Interface Summary 349

Appendix B Keyboard Interface Summary 357

Appendix C Guidelines Summary 361
General Design 361
Design Process 362
Input and Interaction. 362
Windows 362
Controls 363
Integrating with the System 364
User Assistance 364
Visual Design. 365
Sound 365
Accessibility 365
International Users 366
Network Users 366

Appendix D Supporting Windows 95 and Windows NT Version 3.51. 367

Appendix E Localization Word Lists 369

Bibliography 377
General Design 377
Graphic Information Design 378
Usability 378
Object-Oriented Design 378
Accessibility 379
Organizations 380

Glossary 381

Introduction

Welcome to *The Windows Interface Guidelines—A Guide for Designing Software*, an indispensable guide to designing software that runs with the Microsoft®Windows® operating system. The design of your software's interface, more than anything else, affects how a user experiences your product. This guide promotes good interface design and visual and functional consistency within and across Windows-based applications.

What's New

Continuing the direction set by Microsoft OLE, the enhancements in the Windows user interface provide a design evolution from the basic and graphical to the more object oriented—that is, from an application-centered interface to a more data-centered one. In response, developers and designers may need to rethink the interface of their software—the basic components and the respective operations and properties that apply to them. This is important because, from a user's perspective, applications have become less the primary focus and more the engines behind the objects in the interface. Users can now interact with data without having to think about applications, allowing them to better concentrate on their tasks.

When adapting your existing Windows-based software, make certain you consider the following important design aspects.

- Title bar text and icons
- Property sheets
- Transfer model (including drag and drop)
- Pop-up menus
- New controls
- Integration with the system
- Help interface
- OLE embedding and OLE linking
- Visual design of windows, controls, and icons
- Window management
- Presentation of minimized windows

These elements are covered in depth throughout this guide.

How to Use This Guide

This guide is intended for those who are designing and developing Windows-based software. It may also be appropriate for those interested in a better understanding of the Windows environment and the human-computer interface principles it supports. The content of the guide covers the following areas.

- Basic design principles and process—fundamental design philosophy, assumptions about human behavior, design methodology, and concepts embodied in the interface.
- Interface elements—descriptive information about the various components in the interface as well as when and how to use them.
- Design details—specific information about the details of effective design and style when using the elements of the interface.
- Additional information—summaries and tables of topics included in the guide for quick reference, a bibliography of works related to interface design, a comprehensive word list translated into 27 languages to assist in product localization, a glossary of terms defined in the guide, and an index.

This guide focuses on the design and rationale of the user interface. Although an occasional technical reference is included, this guide does not generally cover detailed information about technical implementation or application programming interfaces (APIs), because there are many different types of development tools that you can use to develop software for Windows. The documentation included with the Microsoft® Win32® Software Development Kit (SDK) is one source of information on specific APIs.

How to Apply the Guidelines

This guide promotes visual and functional consistency within and across the Windows operating system. Although following these guidelines is encouraged, you are free to adopt the guidelines that best suit your software. However, you and your customers will benefit if, by following these guidelines, you enable users to transfer their skills and experience from one task to the next and to learn new tasks easily. The data-centered design environment begins to break down the lines between traditional application domains. Inconsistencies in the interface become more obvious and more distracting to users.

Conversely, adhering to the design guidelines does not guarantee usability. The guidelines are valuable tools, but they must be combined with other factors as part of an effective software design process, such factors as application of design principles, task analysis, prototyping, and usability evaluation.

You may extend these guidelines, provided that you do so in the spirit of the principles on which they are based, and maintain a reasonable level of consistency with the visual and behavioral aspects of the Windows interface. In general, avoid adding new elements or behaviors unless the interface does not otherwise support them. More importantly, avoid changing an existing behavior for common elements. A user builds up expectations about the workings of an interface. Inconsistencies not only confuse the user, they also add unnecessary complexity.

These guidelines supersede those issued for Windows version 3.1 and all previous releases and are specific to the development of applications designed for Microsoft® Windows® 95, Microsoft® Windows NT™ Workstation 3.51 (and Microsoft® Windows NT Server 3.51), and later releases. There is no direct relationship between these guidelines and those provided for other operating systems.

Notational Conventions

The following notational conventions are used throughout this guide.

Convention	Used for
SMALL CAPITAL LETTERS	Names of keys on the keyboard—for example, SHIFT, CTRL, or ALT.
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another—for example, CTRL+P or ALT+F4.
KEY,KEY	Key sequences for which the user must press and release one key and then press and release another—for example, ALT,SPACEBAR.
Italic text	New terms and variable expressions, such as parameters.
Bold text	Win32 API keywords.
`Monospace text`	Examples of registry entries.
[ ]	Optional information.

Cross-references to related topics are shown in the margin. Special notes about material are shown in line.

CHAPTER 1

Design Principles and Methodology

A well-designed user interface is built on principles and a development process that centers on users and their tasks. This chapter summarizes the basic principles of the interface design for Microsoft Windows. It also includes techniques and methodologies employed in an effective human-computer interface design process.

User-Centered Design Principles

The information in this section describes the design principles on which Windows and the guidelines in this book are based. You will find these principles valuable when designing software for Windows.

User in Control

An important principle of user interface design is that the user should always feel in control of the software, rather than feeling controlled by the software. This principle has a number of implications.

The first implication is the operational assumption that actions are started not by the computer or the software but by the user, a user who plays an active, rather than reactive, role. Task automation and constraints are still possible, but you should implement them in a balanced way that allows the user freedom of choice.

The second implication is that users, because of their widely varying skills and preferences, must be able to customize aspects of the interface. The system software provides user access to many of these aspects. Your software should reflect user settings for different system properties such as color, fonts, or other options.

The final implication is that your software should be as interactive and responsive as possible. Avoid modes whenever possible. A *mode* is a state that excludes general interaction or otherwise limits the user to specific interactions. When a mode is the only or the best design alternative—for example, for selecting a particular tool in a drawing program—make certain the mode is obvious, visible, the result of an explicit user choice, and easy to cancel.

For information about applying the design principle of user in control, see Chapter 4, "Input Basics," and Chapter 5, "General Interaction Techniques." These chapters cover the basic forms of interaction your software should support.

Directness

Design your software so that users can directly manipulate software representations of information. Whether dragging an object to relocate it or navigating to a location in a document, users should see how the actions they take affect the objects on the screen. Visibility of information and choices also reduce the user's mental workload. Users can recognize a command easier than they can recall its syntax.

Familiar metaphors provide a direct and intuitive interface to user tasks. By allowing users to transfer their knowledge and experience, metaphors make it easier to predict and learn the behaviors of software-based representations.

When using metaphors, you need not limit a computer-based implementation to its "real world" counterpart. For example, unlike its paper-based counterpart, a folder on the Windows desktop can be used to organize a variety of objects such as printers, calculators, and other folders. Similarly, a Windows folder can be more easily resorted. The purpose of using metaphor in the interface is to provide a cognitive bridge; the metaphor is not an end in itself.

Metaphors support user recognition rather than recollection. Users remember a meaning associated with a familiar object easier than they remember the name of a particular command.

For information about applying the principle of directness and metaphor, see Chapter 5, "General Interaction Techniques," and Chapter 13, "Visual Design." These chapters cover, respectively, the use of directness in the interface (including drag and drop) and the use of metaphors when designing icons or other graphical elements.

Consistency

Consistency allows users to transfer existing knowledge to new tasks, learn new things more quickly, and focus more on tasks because they need not spend time trying to remember the differences in interaction. By providing a sense of stability, consistency makes the interface familiar and predictable.

Consistency is important through all aspects of the interface, including names of commands, visual presentation of information, and operational behavior. To design consistency into software, you must consider several aspects.

- Consistency within a product. Present common functions using a consistent set of commands and interfaces. For example, do not provide a Copy command that immediately carries out an operation in one situation but in another presents a dialog box that requires a user to type in a destination. As a corollary to this example, use the same command to carry out functions that seem similar to the user.
- Consistency within the operating environment. By maintaining a high level of consistency between the interaction and interface conventions provided by
Windows, your software benefits from users' ability to apply interaction skills they have already learned.
- Consistency with metaphors. If a particular behavior is more characteristic of a different object than its metaphor implies, the user may have difficulty learning to associate that behavior with an object. For example, an incinerator communicates a different model than a wastebasket for the recoverability of objects placed in it.

Although applying the principle of consistency is the primary goal of this guide, the following chapters focus on the elements common to all Windows-based software: Chapter 6, "Windows," Chapter 7, "Menus, Controls, and Toolbars," and Chapter 8, "Secondary Windows." For information about closely integrating your software with the Windows environment, see Chapter 10, "Integrating with the System," and Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Forgiveness

Users like to explore an interface and often learn by trial and error. An effective interface allows for interactive discovery. It provides only appropriate sets of choices and warns users about potential situations where they may damage the system or data, or better, makes actions reversible or recoverable.

Even within the best designed interface, users can make mistakes. These mistakes can be both physical (accidentally pointing to the wrong command or data) and mental (making a wrong decision about which command or data to select). An effective design avoids situations that are likely to result in errors. It also accommodates potential user errors and makes it easy for the user to recover.

For information about applying the principle of forgiveness, see Chapter 12, "User Assistance," which provides information on supporting discoverability in the interface through the use of contextual, task-oriented, and reference

forms of user assistance. For information about designing for the widest range of users, see Chapter 14, "Special Design Considerations."

Feedback

Always provide feedback for a user's actions. Visual, and sometimes audio, cues should be presented with every user interaction to confirm that the software is responding to the user's input and to communicate details that distinguish the nature of the action.

Effective feedback is timely, and is presented as close to the point of the user's interaction as possible. Even when the computer is processing a particular task, provide the user with information regarding the state of the process and how to cancel that process if that is an option. Nothing is more disconcerting than a "dead" screen that is unresponsive to input. A typical user will tolerate only a few seconds of an unresponsive interface.

It is equally important that the type of feedback you use be appropriate to the task. Pointer changes or a status bar message can communicate simple information; more complex feedback may require the display of a message box.

For information about applying the principle of visual and audio feedback, see Chapter 13, "Visual Design," and Chapter 14, "Special Design Considerations."

Aesthetics

The visual design is an important part of a software's interface. Visual attributes provide valuable impressions as well as communicate important cues to the interaction behavior of particular objects. At the same time, it is important to remember that every visual element that appears on the screen potentially competes for the user's attention. Provide a pleasant environment that clearly contributes to the user's understanding of the information presented. A graphics or visual designer may be invaluable with this aspect of the design.

For information and guidelines related to the aesthetics of your interface, see Chapter 13, "Visual Design." This chapter covers everything from individual element design to font use and window layout.

Simplicity

An interface should be simple (not simplistic), easy to learn, and easy to use. It must also provide access to all functionality provided by an application. Maximizing functionality and maintaining simplicity work against each other in the interface. An effective design balances these objectives.

One way to support simplicity is to reduce the presentation of information to the minimum required to communicate adequately. For example, avoid wordy descriptions for command names or messages. Irrelevant or verbose phrases clutter your design, making it difficult for users to easily extract essential information. Another way to design a simple but useful interface is to use natural mappings and semantics. For example, arranging elements together strengthens their association.

You can also help users manage complexity by using progressive disclosure. *Progressive disclosure* involves careful organization of information so that it is shown only at the appropriate time. By "hiding" information presented to the user, you reduce the amount of information to process. For example, clicking a menu displays its choices; the use of dialog boxes can reduce the number of menu options.

Progressive disclosure does not imply using unconventional techniques for revealing information, such as requiring a modifier key as the only way to access basic functions or forcing the user down a longer sequence of hierarchical interaction. This can make an interface more complex and cumbersome.

For information about applying the principle of simplicity, see Chapter 7, "Menus, Controls, and Toolbars." This chapter discusses progressive disclosure in detail and describes how and when to use the standard (system-supplied) elements in your interface.

Design Methodology

Effective interface design is more than just following a set of rules. It requires a user-centered attitude and design methodology. It also involves early planning of the interface and continued work through the software development process.

A Balanced Design Team

An important consideration in the design of a product is the composition of the team that designs and builds it. Always try to balance disciplines and skills, including development, visual design, writing, human factors, and usability assessment. Rarely are these characteristics found in a single individual, so create a team of individuals who specialize in these areas and who can contribute uniquely to the final design.

Ensure that the design team can effectively work and communicate together. Locating them in the same area of the building or office space, or providing them with a common area to work out design details fosters better communication and interaction.

The Design Cycle

An effective user-centered design process involves a number of important phases: designing, prototyping, testing, and iterating. The following sections describe these phases.

Design

The initial work on a software's design can be the most critical because, during this phase, you decide the general shape of your product. If the foundation work is flawed, it is difficult to correct afterwards.

This part of the process involves not only defining the objectives and features for your product, but understanding who your users are and their tasks, intentions, and goals. This includes understanding factors such as their background—age, gender, expertise, experience level, physical limitations, and special needs; their work environment—equipment, social and cultural influences, and physical surroundings; and their current task organization—the steps required, the dependencies, redundant activities, and the output objective. An order-entry system may have very different users and requirements than an information kiosk.

At this point, begin defining your conceptual framework to represent your product with the knowledge and experience of your target audience. Ideally, you want to create a design model that fits the user's conceptual view of the tasks to be performed. Consider the basic organization and different types of metaphors that can be employed. Often, observing users at their current tasks can provide ideas on effective metaphors to use.

Document your design. Committing your planned design to a written format not only provides a valuable reference point and form of communication, but often helps make the design more concrete and reveals issues and gaps.

Prototype

After you have defined a design model, prototype some of the basic aspects of the design. This can be done with "pencil and paper" models—where you create illustrations of your interface to which other elements can be attached; storyboards—comic book-like sequences of sketches that illustrate specific processes; animations—movie-like simulations; or operational software using a prototyping tool or normal development tools.

A prototype is a valuable asset in many ways. First, it provides an effective tool for communicating the design. Second, it can help you define task flow and better visualize the design. Finally, it provides a low-cost vehicle for getting user input on a design. This is particularly useful early in the design process.

The type of prototype you build depends on your goal. Functionality, task flow, interface, operation, and documentation are just some of the different aspects of a product that need to be assessed. For example, pen and paper models or storyboards may work when defining task organization or conceptual ideas. Operational prototypes are usually best for the mechanics of user interaction.

Consider whether to focus your prototype on breadth or depth. The broader the prototype, the more features you should try to include to gain an understanding about how users react to concepts and organization. When your objective is focused more on detailed usage of a particular feature or area of the design, use depth-oriented prototypes that include more detail for a given feature or task.

Test

User-centered design involves the user in the design process. Usability testing a design, or a particular aspect of a design, provides valuable information and is a key part of a product's success. Usability testing is different than quality assurance testing in that, rather than find programming defects, you assess how well the interface fits user needs and expectations. Of course, defects can sometimes affect how well the interface will fit.

Usability testing provides you not only with task efficiency and success-or-failure data, it also can provide you with information about the user's perceptions, satisfaction, questions, and problems, which may be just as significant as the ability to complete a particular task.

When testing, it is important to use participants who fit the profile of your target audience. Using fellow workers from down the hall might be a quick way to find participants, but software developers rarely have the same experience as their customers. The following section, "Usability Assessment in the Design Process," provides details about conducting a usability test.

There can be different reasons for testing. You can use testing to look for potential problems in a proposed design. You can also focus on comparative studies of two or more designs to determine which is better, given a specific task or set of tasks.

Iterate

Because testing often uncovers design weaknesses, or at least provides additional information you will want to use, repeat the entire process, taking what you have learned and reworking your design or moving onto reprototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results.

During this iterative process, you can begin substituting the actual application for prototypes as the application code becomes available. However, avoid delaying your design cycle waiting for the application code to be complete enough; you can lose valuable time and input that you could have captured with a prototype. Moreover, by the time most applications are complete enough for testing, it is difficult to consider significant changes. This happens for two reasons: 1) it becomes easier to ignore usability defects because of the time and resources invested, and 2) it usually delays the application's delivery schedule.

Usability Assessment in the Design Process

As described in the previous section, usability testing is a key part of the design process, but testing design prototypes is only one part of the picture. Usability assessment begins in the early stages of product development, where you can use it to gather data about how users do their work. You then roll your findings back into the design process. As the design progresses, usability assessment continues to provide valuable input for analyzing initial

design concepts and, in the later stages of product development, can be used to test specific product tasks. Apply usability assessment early and often.

Consider the user's entire experience as part of a product's usability. The usability assessment should include all of a product's components. A software interface is more than just what shows up on the screen or in the documentation.

Usability Testing Techniques

Usability testing involves a wide range of techniques and investment of resources, including trained specialists working in sound-proofed labs with one-way mirrors and sophisticated recording equipment. However, even the simplest investment of an office or conference room, tape recorder, stopwatch, and notepad can produce benefits. Similarly, all tests need not involve great numbers of subjects. More typically, quick, iterative tests with a small, well-targeted sample, 6–10 participants, can identify 80 to 90 percent of most design problems. You can achieve that level with as few as 3–4 users if you only target a single skill level of users, such as novices or immediate level users.

Like the design process itself, usability testing begins with defining the target audience and test goals. When designing a test, focus on tasks, not features. Even if your goal is testing specific features, remember that your customers will use them within the context of particular tasks. It is also a good idea to run a pilot test to work out the bugs of the tasks to be tested and make certain the task scenarios, prototype, and test equipment work smoothly.

When conducting the usability test, provide an environment comparable to the target setting; usually a quiet location, free from distractions, is best. Make participants feel comfortable. It often helps to emphasize that you are testing the software, not the participants. If they become confused or frustrated, it is not a reflection upon them. Unless you have participated yourself, you may be surprised by the pressure many test participants feel. You can alleviate some pressure by explaining the testing process and equipment to the participants.

Allow the user reasonable time to try and work through a difficult situation they encounter. Although it is generally best to not interrupt participants during a test, they may get stuck or end up in situations that require intervention. This need not necessarily disqualify the test data, as long as the test coordinator carefully guides or hints around a problem. Begin with general hints before moving to specific advice. For more difficult situations, you may need to stop the test and make adjustments; Keep in mind that less intervention usually yields better results. Always record the techniques and search patterns that users employ when attempting to work through a difficulty, and the number and type of hints you have to provide them.

Ask subjects to think aloud as they work, so you can hear what assumptions and inferences they are making. As the participants work, record the time they take to perform a task as well as any problems they encounter. You may also want to follow up the session with a questionnaire that asks the participants to evaluate the product or tasks they performed.

Record the test results using a portable tape recorder, or better, a video camera. Since even the best observer can miss details, reviewing the data later will prove invaluable. Recorded data also allows more direct comparisons between multiple participants. It is usually risky to base conclusions on observing a single subject. Recorded data also allows all the design team to review and evaluate the results.

Whenever possible, involve *all* members of the design team in observing the test and reviewing the results. This ensures a common reference point and better design solutions as team members apply their own insights to what they observe. If direct observation is not possible, make the recorded results available to the entire team.

Other Assessment Techniques

There are many techniques you can use to gather usability information. In addition to those already mentioned, "focus groups" are helpful for generating initial ideas or trying out ideas. A focus group requires a moderator who directs the discussion about aspects of a task or design, but allows participants to freely express their opinions. You can also conduct demonstrations, or "walkthroughs," in which you take the user through a set of sample scenarios and ask about their impressions along the way. In a so-called "Wizard of Oz" technique, a testing specialist simulates the interaction of an interface. Although these latter techniques can be valuable, they often require a trained, experienced test coordinator.

Understanding Users

The design and usability techniques described in the previous sections have been used in the development of Windows and in many of the guidelines included in this book. That process has yielded the following general characteristics about users. Consider these characteristics in the design of your software.

- Beginning Windows users often have difficulty with the mouse. For example, dragging and double-clicking are skills that may take time for beginning mouse users to master. Dragging may be difficult because it requires continued pressure on the mouse button and involves properly targeting the correct destination. Double-clicking is not the same as two separate clicks, so many beginning users have difficulty handling the timing necessary to distinguish these two actions, or they overgeneralize the behavior to assume that everything needs double-clicking. Design your interface so that double-clicking and dragging are not the only ways to perform basic tasks; allow the user to conduct the task using single click operations.

- Beginning users often have difficulty with window management. They do not always realize that overlapping windows represent a three-dimensional space. As a result, when a window is hidden by another, a user may assume it no longer exists.

- Beginning users often have difficulty with file management. The organization of files and folders nested more than two levels is more difficult to understand because it is not as obvious in the real world.

- Intermediate users may understand file hierarchies, but have difficulty with other aspects of file management—such as moving and copying files. This may be because most of their experience working with files is often from within an application.

- Advanced, or "power," users want efficiency. The challenge in designing for advanced users is providing for efficiency without introducing complexity for less-experienced users. (Shortcut methods are often useful for supporting these users.) In addition, advanced users may be dependent upon particular interfaces, making it difficult for them to adapt to significant rearrangement or changes in an interface.

- To develop for the widest audience, consider international users and users with disabilities. Including these users as part of your planning and design cycle is the best way to ensure that you can accommodate them.

Design Tradeoffs

A number of additional factors may affect the design of a product. For example, competition may require you to deliver a product to market with a minimal design process, or comparative evaluations may force you to consider additional features. Remember that additional features and shortcuts can affect the product. There is no simple equation to determine when a design tradeoff is appropriate. So in evaluating the impact, consider the following.

- Every additional feature potentially affects performance, complexity, stability, maintenance, and support costs of an application.
- It is harder to fix a design problem after the release of a product because users may adapt, or even become dependent on, a peculiarity in the design.

- Simplicity is not the same as being simplistic. Making something simple to use often requires a good deal of work and code.
- Features implemented by a small extension in the application code do not necessarily have a proportional effect in a user interface. For example, if the primary task is selecting a single object, extending it to support selection of multiple objects could make the frequent, simple task more difficult to carry out.

CHAPTER 2

Basic Concepts

Microsoft Windows supports the evolution and design of software from a basic graphical user interface to a data-centered interface that is better focused on users and their tasks. This chapter outlines the fundamental concepts of data-centered design. It covers some of the basic definitions used throughout this guide and provides the fundamental model for how to define your interface to fit well within the Windows environment.

Data-Centered Design

Data-centered design means that the design of the interface supports a model where a user can browse for data and edit it directly instead of having to first locate an appropriate editor or application. As a user interacts with data, the corresponding commands and tools to manipulate the data or the view of the data automatically become available to the user. This frees a user to focus on the information and tasks rather than on applications and how applications interact.

In this data-centered context, a *document* is a common unit of data used in tasks and exchanged between users. The use of the term document is not limited to the output of a word-processing or spreadsheet application, however. The emphasis is on the data, not the software.

Objects as Metaphor

A well-designed user interface provides an understandable, consistent framework in which users can work, without being confounded by the details of the underlying technology. To help accomplish this, the design model of the Windows user interface uses the metaphor of objects. This is a natural way we interpret and interact with the world around us. In the interface, *objects* not only describe files or icons, but any unit of information, including cells, paragraphs, characters, and circles, and the documents in which they reside.

Object Characteristics

Objects, whether real-world or computer representations, have certain characteristics that help us understand what they are and how they behave. The following concepts describe the aspects and characteristics of computer representations:

- Properties — Objects have certain characteristics or attributes, called *properties*, that define their appearance or state—for example, color, size, and modification date. Properties are not limited to the external or visible traits of an object. They may reflect the internal or operational state of an object, such as an option in a spelling check utility that automatically suggests alternative spellings.
- Operations — Things that can be done with or to an object are considered its *operations*. Moving or copying an object are examples of operations. You can expose operations in the interface through a variety of mechanisms, including commands and direct manipulation.
- Relationships — Objects always exist within the context of other objects. The context, or *relationships*, that an object may have often affects the way the object appears or behaves. The most common relationships are collection, constraint, and composite.

Relationships

The simplest relationship is *collection*, in which objects in a set share a common aspect. The results of a query or a multiple selection of objects are examples of a collection. The significance of a collection is that it enables operations to be applied to the set.

A *constraint* is a stronger relationship between a set of objects in that changing an object in the set affects some other object in the set. The way a text box streams text, the way a drawing application layers its objects, and even the way a word-processing application organizes a document into pages are all examples of constraints.

When a relationship between objects becomes so significant that the aggregation can be identified as an object itself with its own set of properties and operations, the relationship is called a *composite*. A range of cells, a paragraph, and a grouped set of drawing objects are examples of composites.

Another common kind of relationship found in the interface is containment. A *container* is an object that is the place where other objects exist, such as text in a document or documents in a folder. A container often influences the behavior of its content. It may add or suppress certain properties or operations of an object placed in it. In addition, a container controls access to its content as well as what kind of object it will accept as its content. This may affect the results when transferring objects from one container to another.

All these aspects contribute to an object's *type*, a descriptive way of distinguishing or classifying objects. Objects of a common type have similar traits and behaviors.

Composition

As in the natural world, the metaphor of objects implies a constructed environment. Objects are compositions of other objects. You can define most tasks supported by applications as a specialized combination or set of relationships between objects. A text document is a composition of text, paragraphs, footnotes, or other items. A table is a combination of cells, a chart, or a particular organization of graphics. When you define user interaction with objects to be as consistent as possible at any level, you can produce complex constructions while maintaining a small, basic set of conventions. These conventions can apply throughout the interface, increasing ease of use. In addition, using composition to model tasks encourages modular, component-oriented design. This allows objects to be potentially adapted or recombined for other uses.

Persistence

In the natural world, objects persist in their existing state unless changed or destroyed. When you use a pen to write a note, you need not invoke a command to ensure that the ink is preserved on the paper. The act of writing implicitly preserves the information. This is the long term direction for objects in the interface as well. Although it is still appropriate to design software that requires explicit user actions to preserve data, consider whether data can be preserved automatically. In addition, view state information, such as cursor position, scroll position, and window size and location, should be preserved so it can be restored when an object's view is reopened.

Putting Theory into Practice

Using objects in an interface design does not guarantee usability. But applying object-based concepts does offer greater potential for a well-designed interface. As with any good user interface design, a good user-centered design process ensures the success and quality of the interface.

The first step to object-based design should begin as any good design with a thorough understanding of what users' objectives and tasks are. When doing the task analysis, identify the basic components or objects used in those tasks and the behavior and the characteristics that differentiate each kind of object, including the relationships of the objects to each other and to the user. Also identify the actions that are performed, the objects to which they apply, and the state information or attributes that each object in the task must preserve, display, and allow to be edited.

Once the analysis is complete, you can start identifying the user interfaces for the objects. Define how the objects you identified are to be presented, either as icons or data elements in a form. Use icons primarily for representing composite or container objects that need to be opened into their own windows. Attribute or state information should typically be presented as properties of the associated object, most often using property sheets. Map behaviors and

operations to specific kinds of interaction, such as menu commands, direct manipulation, or both. Make these accessible when the object is selected by the user. The information in this guide will help you define how to apply the interfaces provided by the system.

Porting an existing Windows 3.1-based application to a more data-centered interface need not require an immediate, complete overhaul. You can begin the evolution by adding contextual interfaces such as pop-up menus, property sheets, and OLE drag-and-drop and by following the recommendations for designing your window title bars and icons.

CHAPTER 3

The Windows Environment

This chapter provides a brief overview of some of the basic elements included in the Microsoft Windows operating system that allow the user to control the environment (sometimes collectively referred to as the *shell*). These elements provide not only the backdrop for a user's environment, but can be landmarks for the user's interaction with your application as well.

The Desktop

The *desktop* represents a user's primary work area; it fills the screen and forms the visual background for all operations. However, the desktop is more than just a background. It can also be used as a convenient location to place objects that are stored in the file system. In addition, for a computer connected to a network, the desktop also serves as a private work area through which a user can still browse and access objects remotely located on the network.

The Taskbar

The taskbar, as shown in Figure 3.1, is a special component of the desktop that can be used to switch between open windows and to access global commands and other frequently used objects. As a result, it provides a home base—an operational anchor for the interface.

Figure 3.1 The taskbar

Like most toolbars, the taskbar can be configured. For example, a user can move the taskbar from its default location and relocate it along another edge of the screen. The user can also configure display options of the taskbar.

The taskbar can provide the user access to your application. It can also be used to provide status information even when your application is not active. Because the taskbar is an interface shared across applications, be sure to follow the conventions and guidelines covered in this guide.

For more information about integrating your application with the taskbar, see Chapter 10, "Integrating with the System."

The Start Button

The Start button at the left side of the taskbar displays a special menu that includes commands for opening or finding files. The Program menu entry automatically includes the Program Manager entries when the system is installed over Windows 3.1. When installing your Windows application, you also can include an entry for your application by placing a shortcut icon in the system's Programs folder.

Window Buttons

Whenever the user opens a primary window, a button is placed in the taskbar for that window. This button provides the user access to the commands of that window and a convenient interface for switching to that window. The taskbar automatically adjusts the size of the buttons to accommodate as many buttons as possible. When the size of the button requires that the window's title be abbreviated, the taskbar also automatically supplies a tooltip for the button.

When a window is minimized, the window's button remains in the taskbar, but is removed when the window is closed.

Taskbar buttons can also be used as drag and drop destinations. When the user drags over a taskbar button, the system activates the associated window, allowing the user to drop within that window.

For more information about drag and drop, see Chapter 5, "General Interaction Techniques."

The Status Area

On the opposite side of the taskbar from the Start menu is a special status area. Your application can place special status or notification indicators here, even when it is not active.

Icons

Icons may appear on the desktop and in windows. *Icons* are pictorial representations of objects. This is different than the use of icons in Windows 3.1, which also represented minimized windows. Your software should provide and register icons for its application file and any of its associated document or data files.

For more information about the use of icons, see Chapter 10, "Integrating with the System." For information about the design of icons, see Chapter 13, "Visual Design."

Windows includes a number of icons that represent basic objects, such as the following.

Table 3.1 Icons

Icon	Type	Function
My Computer	Computer	Provides access to a user's private storage.
Network Neighborhood	Network	Provides access to the network.
Folder	Folder	Provides organization of files and folders.

Table 3.1 Icons (*continued*)

Icon	Type	Function
	Shortcut	Provides access to other objects. (Typically, shortcut icons are links used for providing convenient access to objects that may be stored elsewhere.)
	Saved Search	Locates files or folders.
	Windows Explorer	Allows browsing of the content of a user's computer or the network.
	Recycle Bin	Stores deleted icons.
	Control Panel	Provides access to properties of installed devices and resources (for example, fonts, displays, and keyboards).

Windows

You can open icons into windows. Windows provides a means of viewing and editing information, and viewing the content and properties of objects. You can also use windows to display parameters to complete commands, palettes of controls, or messages informing a user of a particular situation. Figure 3.2 demonstrates some of the different uses for windows.

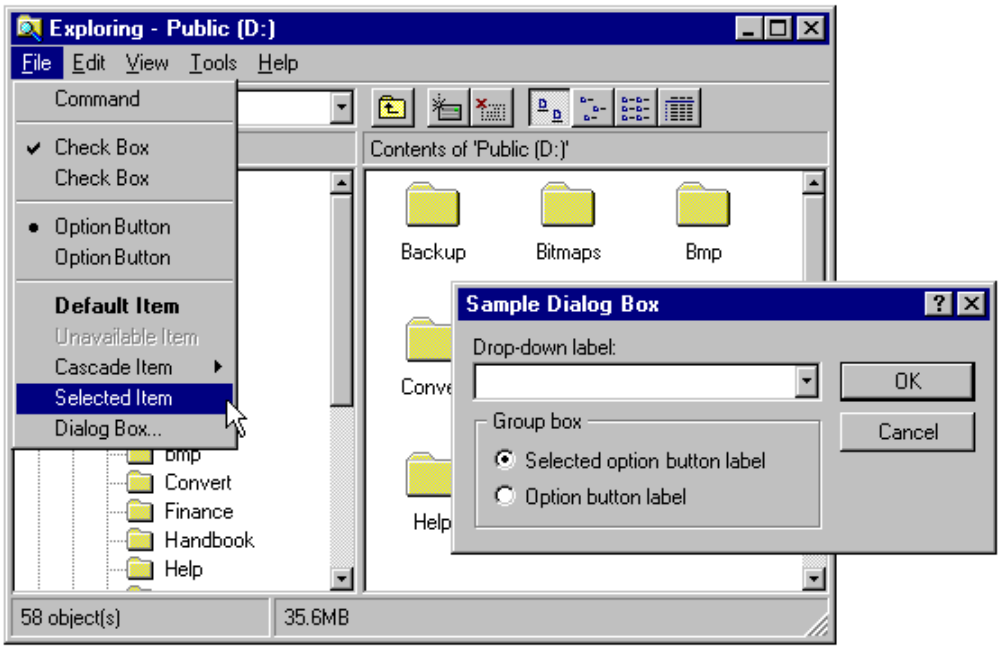


Figure 3.2 Different uses of windows

For more information about windows, see Chapter 6, "Windows," and Chapter 8, "Secondary Windows."

CHAPTER 4

Input Basics

A user can interact with objects in the interface using different types of input devices. The most common input devices are the mouse, the keyboard, and the pen. This chapter covers the basic behavior for these devices; it does not exclude other forms of input.

Mouse Input

The mouse is a primary input device for interacting with objects in the Microsoft Windows interface. Other types of pointing devices that emulate a mouse, such as trackballs, fall under the general use of the term "mouse."

For more information about interactive techniques such as navigation, selection, viewing, editing, transfer, and creating new objects, see Chapter 5, "General Interaction Techniques."

Mouse Pointers

The mouse is operationally linked with a graphic on the screen called the *pointer* (also referred to as the *cursor*). By positioning the pointer and clicking the buttons on the mouse, a user can select objects and their operations.

As a user moves the pointer across the screen, its appearance can change to provide feedback about a particular location, operation, or state. Table 4.1 lists common pointer shapes and their uses.

Table 4.1 Common Pointers

Shape	Screen location	Indicates available or current action
	Over most objects	Pointing, selecting, moving, resizing.
	Over text	Selecting text.
	Over any object or location	Processing an operation.
	Over any screen location	Processing in the background (application loading), but the pointer is still interactive.
	Over most objects	Contextual Help mode.
	Inside a window	Zooming a view.
	Along column gridlines	Resizing a column.
	Along row gridlines	Resizing a row.
	Over split box in vertical scroll bar	Splitting a window (or adjusting a split) horizontally.
	Over split box in horizontal scroll bar	Splitting a window (or adjusting a split) vertically.

	Over any object	Not available.

Your software can define additional pointers, as needed.

Each pointer has a particular point—called a *hot spot*—that defines the exact screen location of the mouse. The hot spot determines what object is affected by mouse actions. Screen objects can additionally define a hot zone; the *hot zone* defines the area the hot spot must be within to be considered over the object. Typically, the hot zone coincides with the borders of an object, but it may be larger, or smaller, to make user interaction easier.

Mouse Actions

All basic mouse actions in the interface use either mouse button 1 or button 2. By default, button 1 is the leftmost mouse button and button 2 is the rightmost button. The system allows the user to swap the mapping of the buttons.

Note For a mouse that supports three buttons, button 2 is the *rightmost* button, not the center button.

The following are the common behaviors performed with the mouse.

Action	Description
Pointing	Positioning the pointer so it "points to" a particular object on the screen without using the mouse button. Pointing is usually part of preparing for some other interaction, because the mouse pointing action is often an opportunity to provide visual cues or other feedback to a user.
Clicking	Positioning the pointer over an object and then pressing and releasing the mouse button. Generally, the mouse is not moved during the click, and the mouse button is quickly released after it is pressed. Clicking identifies (selects) or activates objects.
Double-clicking	Positioning the pointer over an object and pressing and releasing the mouse button twice in rapid succession. Double-clicking an object typically invokes its default operation.
Pressing	Positioning the pointer over an object and then holding down the mouse button. Pressing is often the beginning of a click or drag operation.
Dragging	Positioning the pointer over an object, pressing down the mouse button while holding the mouse button down, and moving the mouse. Use dragging for actions such as selection and direct manipulation of an object.

For most mouse interactions, pressing the mouse button only identifies an operation. User feedback is usually provided at this point. Releasing the mouse button activates (carries out) the operation. An operation that automatically repeats is an exception—for example, pressing a scroll arrow.

This guide does not cover other mouse behaviors such as *chording* (pressing multiple mouse buttons simultaneously) and multiple-clicking (triple- or quadruple-clicking). Because these behaviors require more user skill, they are not generally recommended for basic operations.

Because not all mouse users have a third button, there is no basic action defined for a third (generally the middle) mouse button. It is best to limit the assignment of operations to this button to those environments where the availability of a third mouse button can be assumed, and for providing redundant or shortcut access to operations adequately supported elsewhere in the interface. When assigning actions to the button, you need to define the behaviors for the actions already described (pointing, clicking, dragging, and double-clicking) for this button.

Keyboard Input

The keyboard is a primary means of entering or editing text information. However, the Windows interface also uses keyboard input to navigate, toggle modes, modify input, and, as a shortcut, to invoke certain operations.

For more information about navigation, toggling modes, modifying input, shortcuts, and selection, see Chapter 5, "General Interaction Techniques."

Following are the common interactive behaviors performed with the keyboard.

Action	Description
Pressing	Pressing and releasing a key. Unlike mouse interaction, keyboard interaction occurs upon the down transition of the key. Pressing typically describes the keyboard interaction for invoking particular commands or for navigation.
Holding	Pressing and holding down a key. Holding typically describes interaction with keys such as ALT, SHIFT, and CTRL that modify the standard behavior of other input—for example, another key press or mouse action.
Typing	Typing input of text information from the keyboard.

Text Keys

Text keys include the following:

- Alphanumeric keys (a–z, A–Z, 0–9)
- Punctuation and symbol keys
- TAB and ENTER keys
- The SPACEBAR

In text-entry contexts, pressing a text key enters the corresponding character and typically displays that character on the screen. Except in special views, the characters produced by the TAB and ENTER keys are not usually visible.

Note These keys can also be used for navigation or for invoking specific operations.

Most keyboards include two keys labeled ENTER: one on the main keyboard, one on the numeric keypad. Because these keys have the same label (and on some keyboards the latter may not be available), assign both keys the same functionality.

Access Keys

An access key is an alphanumeric key—sometimes referred to as a *mnemonic*—that when used in combination with the ALT key navigates to and activates a control. The access key matches one of the characters in the text label of the control. For example, pressing ALT+O activates a control whose label is "Open" and whose assigned access key is "O". Typically, access keys are not case sensitive. The effect of activating a control depends on the type of control.

Assign access key characters to controls using the following guidelines (in order of choice):

1. The first letter of the label for the control, unless another letter provides a better mnemonic association.
2. A distinctive consonant in the label.
3. A vowel in the label.

Nonunique access key assignments within the same scope access the first control. Depending on the control, if the user presses the access key a second time, it may or may not access another control with the same assignment. Therefore, define an access key to be unique within the scope of its interaction—that is, the area in which the control exists and to which keyboard input is currently being directed.

Controls without explicit labels can use static text to create labels with assigned access keys. Software that supports a nonroman writing system (such as Kanji), but that runs on a standard keyboard, can prefix each control label with an alphabetic (roman) character as its access key.

For more information about static text controls, see Chapter 7, "Menus, Controls, and Toolbars."

Mode Keys

Mode keys change the actions of other keys (or other input devices). There are two kinds of mode keys: toggle keys and modifier keys.

A toggle key turns a particular mode on or off each time it is pressed. For example, pressing the CAPS LOCK key toggles uppercase alphabetic keys; pressing the NUM LOCK key toggles between numeric and directional input using the keypad keys.

Modifier keys include the SHIFT, CTRL, and ALT keys. Like toggle keys, modifier keys change the actions of normal input. Unlike toggle keys, however, modifier keys establish modes that remain in effect only while the modifier key is held down. Such a "spring-loaded" mode is often preferable to a "locked" mode because it requires the user to continuously activate it, making it a conscious choice and allowing the user to easily cancel the mode by releasing the key.

Because it can be difficult for a user to remember multiple modifier assignments, avoid using multiple modifier keys as the primary means of access to basic operations. In some contexts, such as pen-input–specific environments, the keyboard may not be available. Therefore, use modifier-based actions only for quick access to operations that are supported adequately elsewhere in the interface.

Shortcut Keys

Shortcut keys (also referred to as accelerator keys) are keys or key combinations that, when pressed, provide quick access to frequently performed operations. CTRL+*letter* combinations and function keys (F1 through F12) are usually the best choices for shortcut keys. By definition, a shortcut key is a keyboard equivalent of functionality that is supported adequately elsewhere in the interface. Therefore, avoid using a shortcut key as the only way to access a particular operation.

When defining shortcut keys, observe the following guidelines:

- Assign single keys where possible because these keys are the easiest for the user to perform.
- Make modified-letter key combinations case insensitive.
- Use SHIFT+*key* combinations for actions that extend or complement the actions of the key or key combination used without the SHIFT key. For example, ALT+TAB switches windows in a top-to-bottom order. SHIFT+ALT+TAB switches windows in reverse order. However, avoid SHIFT+*text* keys, because the effect of the SHIFT key may differ for some international keyboards.
- Use CTRL+*key* combinations for actions that represent a larger scale effect. For example, in text editing contexts, HOME moves to the beginning of a line, and CTRL+HOME moves to the beginning of the text. Use CTRL+*key* combinations for access to commands where a letter key is used—for example, CTRL+B for bold. Remember that such assignments may be meaningful only for English-speaking users.
- Avoid ALT+*key* combinations because they may conflict with the standard keyboard access for menus and controls. The ALT+*key* combinations—ALT+TAB, ALT+ESC, and ALT+SPACEBAR—are reserved for system use. ALT+*number* combinations enter special characters.
- Avoid assigning shortcut keys defined in this guide to other operations in your software. That is, if CTRL+C is the shortcut for the Copy command and your application supports the standard copy operation, don't assign CTRL+C to another operation.
- Provide support for allowing the user to change the shortcut key assignments in your software, when possible.
- Use the ESC key to stop a function in process or to cancel a direct manipulation operation. It is also usually interpreted as the shortcut key for a Cancel button.

Note Function key and modified function key combinations may be easier for international users because they have no mnemonic relationship. However, there is a tradeoff because function keys

are often more difficult to remember and to reach. For a list of the most common shortcut key assignments, see Appendix B, "Keyboard Interface Summary."

Some keyboards also support three new keys, the Application key and the two Windows keys. The primary use for the Application key is to display the pop-up menu for current selection (same as SHIFT+F10). You may also use it with modifier keys for application-specific functions. Pressing either of the Windows keys—left or right—displays the Start button menu. These keys are also used by the system as modifiers for system-specific functions. Do not use these keys as modifiers for non–system-level functions.

For more information about pop-up menus, see Chapter 7, "Menus, Controls, and Toolbars."

Pen Input

Systems with a Windows pen driver installed support user input using tapping or writing on the surface of the screen with a pen, and in some cases with a finger. Your software can determine whether Windows pen extensions have been installed by checking the SM_PENWINDOWS constant using the **GetSystemMetrics** function.

For more information about the SM_PENWINDOWS constant and the **GetSystemMetrics** function, see the *Microsoft Win32 Programmer's Reference*.

Depending on where the pen is placed, you can use it for both pointing and writing. For example, if you move the pen over menus or most controls, it acts as a pointing device. Because of the pointing capabilities of the pen, the user can perform most mouse-based operations. When over a text entry or drawing area, the pen becomes a writing or drawing tool; the pointer changes to a pen shape to provide feedback to the user. When the tip of the pen touches the screen, the pen starts *inking*—that is, tracing lines on the screen. The user can then draw shapes, characters, and other patterns; these patterns remain on the screen exactly as drawn or can be recognized, interpreted, and redisplayed.

The pen can retain the functionality of a pointing device (such as a mouse) even in contexts where it would normally function as a writing or drawing tool. For example, you can use timing to differentiate operations; that is, if the user holds the pen tip in the same location for a predetermined period of time, a different action may be inferred. This method is often unreliable or inefficient for many operations, however, so it may be better to use toolbar buttons to switch to different modes of operation. Choosing a particular button allows the user to define whether to use the pen for entering information (writing or drawing) or as a pointing device.

You can also provide the user with access to other operations using an action handle. An action handle can be used to support direct manipulation operations or to provide access to pop-up menus.

For more information about action handles, see Chapter 7, "Menus, Controls, and Toolbars."

Note When designing for pen input, it is more important to make the interface easy to use than to assume all actions should be based on handwriting recognition or gestural interfaces. Often, the pen can be more effective as a pointing device than as a text-entry device.

Following are the fundamental behaviors defined for a pen.

Action	Description
Pressing	Positioning the pen tip over the screen and pressing the tip to the screen. A pen press is equivalent to a mouse press and typically identifies a particular pen action.
Tapping	Pressing the pen tip on the screen and lifting it without moving the pen. In general, tapping is equivalent to clicking mouse button 1. Therefore, this action typically selects an object, setting a text insertion point or activating a button
Double-tapping	Pressing and lifting the pen tip twice in rapid succession. Double-tapping is usually interpreted as the equivalent to double-clicking mouse button 1.
Dragging	Pressing the pen tip on the screen and keeping it pressed to the screen while moving the pen. In inking contexts, you can use dragging for the input of pen strokes for writing, drawing, gestures, or for direct manipulation, depending on which is most appropriate for the context. In noninking contexts, it is the equivalent of a mouse drag.

Some pens include buttons on the pen barrel that can be pressed. For pens that support barrel buttons, the following behaviors may be supported.

Action	Description
Barrel-tapping	Holding down the barrel button of the pen while tapping. Barrel-tapping is equivalent to clicking with mouse button 2.
Barrel-dragging	Holding down the barrel button of the pen while dragging the pen. Barrel-dragging is equivalent to dragging with mouse button 2.

Note Because not all pens support barrel buttons, any behaviors that you support using a barrel button should also be supported by other techniques in the interface.

Pen input is delimited, either by the lifting of the pen tip, an explicit termination tap (such as tapping the pen on another window or as the completion of a gesture), or a time-out without further input. You can also explicitly define an application-specific recognition time-out.

Proximity is the ability to detect the position of the pen without it touching the screen. While Windows provides support for pen proximity, avoid depending on proximity as the exclusive means of access to basic functions, because not all pen hardware supports this feature. Even pen hardware that does support proximity may allow other non-pen input, such as touch input, where proximity cannot be supported.

Pen Pointers

Because the pen (unlike the mouse) points directly at the screen, graphical onscreen pointers may seem superfluous; however, they do have an important role to play. Pointers help the pen user select small targets faster. Moreover, changes from one pointer to another provide useful feedback about the actions supported by the object under the pen. For example, when the pen moves over a resizable border, the pointer can change from a pen (indicating that writing is possible) to a resizing pointer (indicating that the border can be dragged to resize the object). Whenever possible, include this type of feedback in pen-enabled applications to help users understand the kinds of supported actions.

Following are two common pointers used with the pen.

Table 4.2 Pen Pointers

Shape	Common usage
	Pointing, selecting, moving, and resizing
	Writing and drawing

Because a pointer may be partially obscured by the pen or by the user's hand, consider including other forms of feedback, such as toolbar button states or status bar information.

Pen Gestures

When using the pen for writing, keep in mind that certain ink patterns are interpreted as *gestures*. Using one of these specially drawn symbols invokes a particular operation, such as deleting text, or produces a nonprinting text character, such as a carriage return or a tab. For example, an *X* shape is equivalent to the Cut command. After the system interprets a gesture, the gesture's ink is removed from the display.

For more information about common gestures and their interpretation, see Chapter 5, "General Interaction Techniques."

All gestures include a circular stroke to distinguish them from ordinary characters. Most gestures also operate positionally; in other words, they act upon the objects on which they are drawn. Determining the position of the specific gesture depends on either the area surrounded by the gesture or a single point—the hot spot of the gesture.

Pen gestures usually cannot be combined with ink (writing or drawing actions) within the same recognition sequence. For example, the user cannot draw a few characters, immediately followed by a gesture, followed by more characters.

The rapidity of gestural commands is one of the key advantages of the pen. Do not rely on gestures as the only or primary way to perform commands, however, because gestures require memorization by users. Regard gestures as a quick access, shortcut method for operations adequately supported elsewhere in the interface, such as in menus or buttons. If the pen extensions are installed, you can optionally place a bitmap of the gesture next to the corresponding command (in place of the keyboard shortcut text) to help the user learn particular gestures.

In addition, avoid using gestures when they interfere with common functionality or making operations with parallel input devices, such as the mouse or keyboard, more cumbersome. For example, although writing a character gesture in a list box could be used as a way to scroll automatically within the list, it would interfere with the basic and more frequent user action of selecting an item in the list. A better technique is to provide a text input field where the user can write and, based on the letters entered, scroll the list.

Pen Recognition

Recognition is the interpretation of pen strokes into some standardized form. Consider recognition as a means to an end, not an end in itself. Do not use recognition if it is unnecessary or if it is not the best interface. For example, it may be more effective to provide a control that allows a user to select a date, rather than requiring the user to write it in just so your software can recognize it.

Where it is appropriate to do so, you can improve recognition by using context and constraints. For example, a checkbook application can constrain certain fields to contain only numbers.

Accurate recognition is difficult to achieve, but you can greatly improve your recognition interface by providing a fast, easy means to correct errors. For example, if you allow users to overwrite characters or choose alternatives, they will be less frustrated and find recognition more useful.

Ink Input

In some cases—for example, signatures—recognition of pen input may be unnecessary; the ink is a sufficient representation of information. Ink is a standard data type supported by the Clipboard. Consider supporting ink entries as input wherever your software accepts normal text input, unless the representation of that input needs to be interpreted for other operations, such as searching or sorting.

Targeting

Targeting, or determining where to direct pen input, is an important design factor for pen-enabled software. For example, if the user gestures over a set of objects, which objects should be affected? If the user writes text that spans several writing areas, which text should be placed in which area? In general, you use the context of the input to determine where to apply pen input. More specifically, use the following guidelines for targeting gestures on objects.

- If the user draws the gesture on any part of a selection, apply the gesture to the selection.
- If the user draws the gesture on an object that is not selected, select that object, and the gesture is applied to that object.
- If the user does not draw the gesture on any object or selection, but there is a selection, apply the gesture to that selection.

If none of these guidelines applies, ignore the gesture.

For handwriting, the context also determines where to direct the input. Figure 4.1 demonstrates how the proximity of the text to the text boxes determines the destination of the written text.

Figure 4.1 Targeting Handwritten input

The system's pen services provide basic support for targeting, but your application can also provide additional support. For example, your application can define a larger inking rectangle than the control usually provides. In addition, because your application often knows the type of input to expect, it can use this information to better interpret where to target the input.

CHAPTER 5

General Interaction Techniques

This chapter covers basic interaction techniques, such as navigation, selection, viewing, editing, and creation. Many of these techniques are based on an object-action paradigm in which a user identifies an object and an action to apply to that object. By maintaining these techniques consistently, you enable users to transfer their skills to new tasks.

Where applicable, support the basic interaction techniques for the mouse, keyboard, and pen. When adding or extending these basic techniques, consider how the feature or function can be supported across input devices. Techniques for a particular device need not be identical for all devices. Instead, tailor techniques to optimize the strengths of a particular device. In addition, make it easy for the user to switch between devices so that an interaction started with one device can be completed with another.

Navigation

One of the most common ways of identifying or accessing an object is by navigating to it. The following sections include information about mouse, pen, and keyboard techniques.

Mouse and Pen Navigation

Navigation with the mouse is simple; when a user moves the mouse left or right, the pointer moves in the corresponding direction on the screen. As the mouse moves away from or toward the user, the pointer moves up or down. By moving the mouse, the user can move the pointer to any location on the screen. Pen navigation is similar to mouse navigation, except that the user navigates by moving the pen across the display without touching the screen.

Keyboard Navigation

Keyboard navigation requires a user to press specific keys and key combinations to move the *input focus*—the indication of where the input is being directed—to a particular location. The appearance of the input focus varies by context; in text, it appears as a text cursor or insertion point.

For more information about the display of the input focus, see Chapter 13, "Visual Design."

Basic Navigation Keys

The navigation keys are the four arrow keys and the HOME, END, PAGE UP, PAGE DOWN, and TAB keys. Pressed in combination with the CTRL key, a navigation key increases the movement increment. For example, where pressing RIGHT ARROW moves right one *character* in a text field, pressing CTRL+RIGHT ARROW moves right one *word* in the text field. Table 5.1 lists the common navigation keys and their functions. You can define additional keys for navigation.

Table 5.1 Basic Navigation Keys

Key	Moves cursor to...	CTRL+key moves cursor to...
LEFT ARROW	Left one unit.	Left one (larger) unit.
RIGHT ARROW	Right one unit.	Right one (larger) unit.
UP ARROW	Up one unit or line.	Up one (larger) unit.
DOWN ARROW	Down one unit or line.	Down one (larger) unit.

HOME	Beginning of line.	Beginning of data or file (topmost position).
END	End of line.	End of data or file (bottommost position).
PAGE UP	Up one screen (previous screen, same position).	Left one screen (or previous unit, if left is not meaningful).
PAGE DOWN	Down one screen (next screen, same position).	Right one screen (or next unit, if right is not meaningful).
TAB	Next field. (SHIFT+TAB moves in reverse order).	Next larger field.

Unlike mouse and pen navigation, keyboard navigation typically affects existing selections. Optionally, though, you can support the SCROLL LOCK key to enable scrolling navigation without affecting existing selections. If you do so, the keys will scroll the appropriate increment.

For more information about keyboard navigation in secondary windows such as dialog boxes, see Chapter 8, "Secondary Windows."

Selection

Selection is the primary means by which the user identifies objects in the interface. Consequently, the basic model for selection is one of the most important aspects of the interface.

Selection typically involves an overt action by the user to identify an object. This is known as an *explicit selection*. Once the object is selected, the user can specify an action for the object.

There are also situations where the identification of an object can be derived by inference or implied by context. An *implicit selection* works most effectively where the association of object and action is simple and visible. For example, when the user drags a scroll box, the user establishes selection of the scroll box and the action of moving at the same time. Implicit selection may result from the relationships of a particular object. For example, selecting a character in a text document may implicitly select the paragraph of which the character is a part.

A selection can consist of a single object or multiple objects. Multiple selections can be *contiguous*—where the selection set is made up of objects that are logically adjacent to each other, also known as a *range selection*. A *disjoint selection* set is made up of objects that are spatially or logically separated.

Multiple selections may also be classified as *homogeneous* or *heterogeneous*, depending on the type or properties of the selected objects. Even a homogeneous selection might have certain aspects in which it is heterogeneous. For example, a text selection that includes bold and italic text can be considered homogeneous with respect to the basic object type (characters), but heterogeneous with respect to the values of its font properties. The homogeneity or heterogeneity of a selection affects the access of the operations or properties of the objects in the selection.

Selection Feedback

Always provide visual feedback as a selection is made, so that the user can tell the effect of the selection operation. Display the appropriate selection appearance for each object included in the selection set. The form of selection appearance depends on the object and its context.

For more information about how to visually render the selection appearance of an object, see Chapter 13, "Visual Design." Chapter 11, "Working with OLE Embedded and OLE Linked Objects," also includes information about how the context of an object can affect its selection appearance.

Scope of Selection

The *scope* of a selection is the area, extent, or region in which, if other selections are made, they will be considered part of the same selection set. For example, you can select two document icons in the same folder window. However, the selection of these icons is independent of the selection of the window's scroll bar, a menu, the window itself, or selections made in other windows. So, the selection scope of the icons is the area viewed through that window. Selections in different scopes are independent of each other. The scope of a selection is important because you use it to define the available operations for the selected items and how the operations are applied.

Hierarchical Selection

Range selections typically include objects at the same level. However, you can also support a user's elevating a range selection to the next higher level if it extends beyond the immediate containment of the object (but within the same window). When the user adjusts the range back within the containment of the start of the range, return the selection to the original level. For example, extending a selection from within a cell in a table to the next cell, as shown in Figure 5.1, should elevate the selection from the character level to the cell level; adjusting the selection back within the cell should reset the selection to the character level.

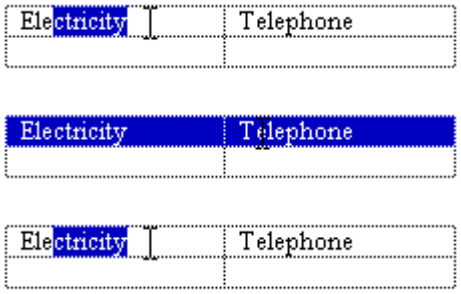


Figure 5.1 Hierarchical selection

Mouse Selection

Selection with the mouse relies on the basic actions of clicking and dragging. In general, clicking selects a single item or location, and dragging selects a single range consisting of all objects logically included from the button-down to the button-up location. If you also support dragging for object movement, use keyboard-modified mouse selection or region selection to support multiple selection.

Selection with the Mouse

Support user selection using either mouse button. When the user presses the mouse button, establish the starting point, or *anchor point*, of a selection. If, while pressing the mouse button, the user drags the mouse, extend the selection to the object nearest the hot spot of the pointer. If, while holding the mouse button down, the user drags the mouse within the selection, reduce the selection to the object now nearest the pointer. Tracking the selection with the pointer while the mouse button continues to be held down allows the user to adjust a range selection dynamically. Use appropriate selection feedback to indicate the objects included in the selection.

For more information about selection feedback appearance, see Chapter 13, "Visual Design."

The release of the mouse button ends the selection operation and establishes the *active end* of the selection. Support selection adjustment with subsequent selection operations using the SHIFT and CTRL keys. If the user presses mouse button 2 to make a selection, display the contextual pop-up menu for the selected objects when the user releases the mouse button.

For more information about pop-up menus, see Chapter 7, "Menus, Controls, and Toolbars."

The most common form of selection optimizes for the selection of a single object or a single range of objects. In such a case, creating a new selection within the scope of an existing selection (for example, within the same area of the window) cancels the selection of the previously selected objects. This allows simple selections to be created quickly and easily.

When using this technique, reset the selection when the user presses the mouse button and the pointer (hot spot) is outside (not on) any existing selection. If the pointer is over a selected item, however, don't cancel the former selection. Instead, determine the appropriate result according to whether the user pressed mouse button 1 or 2.
If the user presses mouse button 1 and the pointer does not move from the button down point, the effect of the release of the mouse button is determined by the context of the selection. You can support whichever of the following best fits the nature of the user's task.

- The result may have no effect on the existing selection. This is the most common and safest effect.
- The object under the pointer may receive some special designation or distinction; for example, become the next anchor point or create a subselection.
- The selection can be reset to be only the object under the pointer.

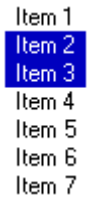
If the user pressed mouse button 2, the selection is not affected, but you display a pop-up menu for selection.

Although selection is typically done by positioning the pointer over an object, it may be inferred based on the logical proximity of an object to a pointer. For example, when selecting text, the user can place the pointer on the blank area beyond the end of the line and the resulting selection is inferred as being the end of the line.

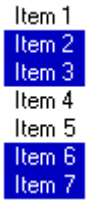
Selection Adjustment

Selections are adjusted (elements added to or removed from the selection) using keyboard modifiers with the mouse. The CTRL key is the disjoint, or toggle, modifier. If the user presses the CTRL key while making a new selection, preserve any existing selection within that scope and reset the anchor point to the new mouse button-down location. Toggle the selection state of the object under the pointer—that is, if it is not selected, select it; if it is already selected, unselect it.

If a selection modified by the CTRL key is made by dragging, the selection state is applied for all objects included by the drag operation (from the anchor point to the current pointer location). This means if the first item included during the drag operation is not selected, select all objects included in the range. If the first item included was already selected, unselect it and all the objects included in the range regardless of their original state. For example, the user selects the first two items in the list by dragging.



The user can then press the CTRL key and drag to create a disjoint selection, resetting the anchor point.



The user must press the CTRL key before using the mouse button for a disjoint (toggle) selection. After a disjoint selection is initiated, it continues until the user releases the mouse button (even if the user releases the CTRL key before the mouse button).
The SHIFT key adjusts (or extends) a single selection or range selection. When the user presses the mouse button while holding down the SHIFT key, reset the active end of a selection from the anchor point to the location of the pointer. Continue tracking the pointer, resetting the active end as the user drags, similar to a simple range drag selection. When the user releases the mouse button, the selection operation ends. You should then set the active end to the object nearest to the mouse button release point. Do not reset the anchor point. It should remain at its current location.

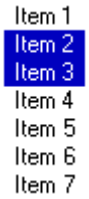
Only the selection made from the current anchor point is adjusted. Any other disjoint selections are not affected unless the extent of the selection overlaps an existing disjoint selection.

The effect on the selection state of a particular object is based on the first item included in the selection range. If the first item is already selected, select (not toggle the selection state of) all objects included in the range; otherwise, unselect (not toggle the selection state of) the objects included.

The user must press and hold down the SHIFT key before pressing the mouse button for the action to be interpreted as adjusting the selection. When the user begins adjusting a selection by pressing the SHIFT key, continue to track the pointer and adjust the selection (even if the user releases the modifier key) until the user releases the mouse button.

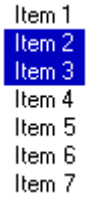
Pressing the SHIFT modifier key always adjusts the selection from the current anchor point. This means the user can always adjust the selection range of a single selection or CTRL key–modified disjoint selection. For example, the user could select the following items by making a range selection by dragging.

The user can accomplish this same result by making an initial selection.



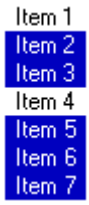
The user can then adjust the selection with the SHIFT key and dragging.

The following sequence illustrates how the user can use the SHIFT key and dragging to adjust a disjoint selection. The user makes the initial selection by dragging.



The user then presses the CTRL key and drags to create a disjoint selection.

The user can then extend the disjoint selection using the SHIFT key and dragging. This adjusts the selection from the anchor point to the button down point and tracks the pointer to the button up point.


Item 1
Item 2
Item 3
Item 4
Item 5
Item 6
Item 7

The following summarizes the mouse selection operations.

Operation	Mouse action
Select object (range of objects)	Click (drag)
Disjoint selection state of noncontiguous object (range of objects)	CTRL+click (drag)
Adjust current selection to object (or range of objects)	SHIFT+click (drag)

For more information about the mouse interface, including selection behavior, see Appendix A, "Mouse Interface Summary."

Region Selection

In Z-ordered, or layered, contexts, in which objects may overlap, user selection can begin on the background (sometimes referred to as *white space*). To determine the range of the selection in such cases, a bounding outline (sometimes referred to as a marquee) is drawn. The outline is typically a rectangle, but other shapes (including freeform outline) are possible.

When the user presses the mouse button and moves the pointer (a form of selection by dragging), display the bounding outline. You set the selection state of objects included by the outline using the selection guidelines described in the previous sections, including operations that use the SHIFT and CTRL modifier keys.

You can use the context of your application and the user's task to determine whether an object must be totally enclosed or only intersected by the bounding region to be affected by the selection operation. Always provide good selection feedback during the operation to communicate to the user which method you support. When the user releases the mouse button, remove the bounding region, but retain the selection feedback.

Pen Selection

When the pen is being used as the pointing device, you can use the same selection techniques defined for the mouse. For example, in text input controls, you support user selection of text by dragging through it. Standard pen interfaces also support text selection using a special pen selection handle. In discrete object scenarios, like drawing programs, you support selection of individual objects by tapping or by performing region selection by dragging.

For more information about selection support in pen-enabled controls, see the "Pen-Specific Editing Techniques" section later in this chapter.

In some contexts, you can also use the *lasso-tap* gesture to support selection of individual objects or ranges of objects. However, avoid implementing this gesture when it might interfere with primary operations such as direct manipulation.

Lasso-tap involves making a circular gesture around the object, then tapping within the gesture. For example, in Figure 5.2, making the lasso-tap gesture selects the word "controversial."

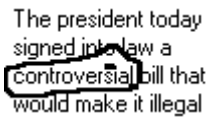


Figure 5.2 A lasso-tap gesture

In text contexts, base the selection on the extent of the lasso gesture and the character-word-paragraph granularity of the text elements covered. For example, if the user draws the lasso around a single character, select only that character. If the user draws the lasso around multiple characters within a word, select the entire word. If the gesture encompasses characters in multiple words, select the range of words logically included by the gesture. This reduces the need for the user to be precise.

Keyboard Selection

Keyboard selection relies on the input focus to define selected objects. The input focus can be an insertion point, a dotted outline box, or some other cursor or visual indication of the location where the user is directing keyboard input.

For more information about input focus, see Chapter 13, "Visual Design."

In some contexts, selection may be implicit with navigation. When the user presses a navigation key, you move the input focus to the location (as defined by the key) and automatically select the object at that location.

In other contexts, it may be more appropriate to move the input focus and require the user to make an explicit selection with the Select key. The recommended keyboard Select key is the SPACEBAR, unless this assignment directly conflicts with the specific context—in which case, you can use CTRL+SPACEBAR. (If this conflicts with your software, define another key that best fits the context.) In some contexts, pressing the Select key may also unselect objects; in other words, it will toggle the selection state of an object.

Contiguous Selection

In text contexts, the user moves the insertion point to the desired location using the navigation keys. Set the anchor point at this location. When the user presses the SHIFT key with any navigation key (or navigation key combinations, such as CTRL+END), set that location as the active end of the selection and select all characters between the anchor point and the active end. (Do not move the anchor point.) If the user presses a subsequent navigation key, cancel the selection and move the insertion point to the appropriate location defined by the key. If the user presses LEFT ARROW or RIGHT ARROW keys, move the insertion point to the end of the former selection range. If UP ARROW or DOWN ARROW are used, move the insertion point to the previous or following line at the same relative location.

You can use this technique in other contexts, such as lists, where objects are logically contiguous. However, in such situations, the selection state of the objects logically included from the anchor point to the active end depend on the selection state of the object at, or first traversed from, the anchor point. For example, if the object at the anchor point is selected, then select all the objects in the range regardless of their current state. If the object at the anchor point is not selected, unselect all the items in the range.

Disjoint Selection

Use the Select key for supporting disjoint selections. The user uses navigation keys or navigation keys modified by the SHIFT key to establish the initial selection. The user can then use navigation keys to move to a new location and subsequently use the Select key to create an additional selection.

In some situations, you may prefer to optimize for selection of a single object or single range. In such cases, when the user presses a navigation key, reset the selection to the location defined by the navigation key. Creating a disjoint

selection requires supporting the Add mode key (SHIFT+F8). In this mode, you move the insertion point when the user presses navigation keys without affecting the existing selections or the anchor point. When the user presses the Select key, toggle the selection state at the new location and reset the anchor point to that object. At any point, the user can use the SHIFT+navigation key combination to adjust the selection from the current anchor point.

When the user presses the Add mode key a second time, you toggle out of the mode, preserving the selections the user created in Add mode. But now, if the user makes any new selections within that selection scope, you return to the single selection optimization—canceling any existing selections—and reset the selection to be only the new selection.

Selection Shortcuts

Double-clicking with mouse button 1 and double-tapping—its pen equivalent—is a shortcut for the default operation of an object. In text contexts, it is commonly assigned as a shortcut to select a word. When supporting this shortcut, select the word and the space following the word, but not the punctuation marks.

> **Note** Double-clicking as a shortcut for selection generally applies to text. In other contexts, it may perform other operations.

You can define additional selection shortcuts or techniques for specialized contexts. For example, selecting a column label may select the entire column. Because shortcuts cannot be generalized across the user interface, however, do not use them as the only way to perform a selection.

Common Conventions for Supporting Operations

There are many ways to support operations for an object, including direct manipulation of the object or its control point (handle), menu commands, buttons, dialog boxes, tools, or programming. Support for a particular technique is not exclusive to other techniques. For example, the user can size a window by using the Size menu command as well as by dragging its border.

Design operations or commands to be *contextual*, or related to, the selected object to which they apply. That is, determine which commands or properties, or other aspects of an object, are made accessible by the characteristics of the object and its context (relationships). Often the context of an object may add to or suppress the traits of the object. For example, the menu for an object may include commands defined by the object's type as well as commands supplied by the object's current container.

Operations for a Multiple Selection

When determining which operations to display for a multiple selection, use an intersection of the operations that apply to the members of that selection. The selection's context may add to or filter out the available operations or commands displayed to the user.

It is also possible to determine the effect of an operation for a multiple selection based upon a particular member of that selection. For example, when the user selects a set of graphic objects and chooses an alignment command, you can make the operation relative to a particular item identified in the selection.

Limit operations on a multiple selection to the scope of the selected objects. For example, deleting a selected word in one window should not delete selections in other windows (unless the windows are viewing the same selected objects).

Default Operations and Shortcut Techniques

An object can have a default operation; a *default operation* is an operation that is assumed when the user employs a shortcut technique, such as double-clicking or drag and drop. For example, double-clicking a folder displays a

window with the content of the folder. For other objects, such as the Mouse object in the Control Panel, double-clicking displays the properties of the object, or, in text editing situations, selects the word. The behavior differs because the default commands in each case differ: for a folder, the default command is Open; for a device object such as a mouse, the command is Properties; and for text, it is Select Word.

Similarly, when the user drags and drops an object at a new location with mouse button 1, there must be a default operation defined to determine the result of the operation. Dragging and dropping to some locations can be interpreted as a move, copy, link, or some other operation. In this case, the drop destination determines the default operation.

For more information about supporting default operations for drag and drop, see the "Transfer Operations" section in this chapter; also see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Shortcut techniques for default operations provide greater efficiency in the interface, an important factor for more experienced users. However, because they typically require more skill or experience and because not all objects may have a default operation defined, avoid shortcut techniques as the exclusive means of performing a basic operation. For example, even though double-clicking opens a folder icon, the Open command appears on its menu.

View Operations

Following are some of the common operations associated with viewing objects. Although these operations may not always be used with all objects, when supported, they should follow similar conventions.

Operation	Action
Open	Opens a primary window for an object. For container objects, such as folders and documents, this window displays the content of the object.
Close	Closes a window.
Properties	Displays the properties of an object in a window, typically in a property sheet window.
Help	Displays a window with the contextual Help information about an object.

When the user opens a new window, you should display it at the top of the Z order of its peer windows and activate it. Primary windows are typically peers. Display supplemental or secondary windows belonging to a particular application at the top of their local Z order— that is, the Z order of the windows of that application, not the Z order of other primary windows.

If the user interacts with another window before the new window opens, the new window does not appear on top; instead, it appears where it would usually be displayed if the user activated another window. For example, if the user opens window A, then opens window B, window B appears on top of window A. If the user clicks back in window A before window B is displayed, however, window A remains active and at the top of the Z order; window B appears behind window A.

Whether opening a window allows the user to also edit the information in that window's view depends on a number of factors. These factors can include who the user is, the type of view being used, and the content being viewed.

After the user opens a window, re-executing the command that opened the window should activate the existing window, instead of opening another instance of the window. For example, if the user chooses the Properties command for an selected object whose property sheet is already open, the existing property sheet is activated, rather than a second window opened.

For more information about opening windows, property sheets, and Help windows, see Chapter 6, "Windows," Chapter 8, "Secondary Windows," and Chapter 12, "User Assistance," respectively.

Note This guideline applies per user desktop. Two users opening a window for the same object on a network can each see separate windows for the object from their individual desktops.

Closing a window does not necessarily mean quitting the processes associated with the object being viewed. For example, closing a printer's window does not cancel the printing of documents in its queue. So even though exiting an application closes its windows, closing a window does not necessarily exit an application. Similarly, you can use other commands in secondary windows which result in closing the window—for example, OK and Cancel. However, the effect of closing the window with a Close command depends on the context of the window. Avoid assuming that the Close command is always the equivalent of the Cancel command.

If there are changes transacted in a window that have not yet been applied and the user chooses the Close command, and those changes will be lost if not applied, display a message asking whether the user wishes to apply or discard the changes or cancel the Close operation. If there are no outstanding changes or if pending changes are retained for the next time the window is opened, remove the window.

View Shortcuts

Following are the recommended shortcut techniques for the common viewing commands.

Shortcut	Operation
CTRL+O	Opens a primary window for an object. For container objects, such as folders and documents, this window displays the content of the object.
ALT+F4	Closes a window.
F1	Displays a window with contextual Help information.
SHIFT+F1 Starts context-sensitive Help mode.	Starts context-sensitive Help mode.
Double-click (button 1) or ENTER Carries out the default command.	Carries out the default command.
ALT+double-click or ALT+ENTER	Displays the properties of an object in a window, typically in a property sheet window.

For more information on reserved and recommended shortcut keys, see Appendix B, "Keyboard Interface Summary."

Use double-clicking and the ENTER key to open a view of an object when that view command is the default command for the object. For example, double-clicking a folder opens the folder's primary window. But double-clicking a mouse object displays its property sheet; this is because the Open command is the default command for folders, and the Properties command is the default command for device objects.

Editing Operations

Editing involves changing (adding, removing, replacing) some fundamental aspect about the composition of an object. Not all changes constitute editing of an object, though. For example, changing the view of a document to an outline or magnified view (which has no effect on the content of the document) is not editing. The following sections cover some of the common interface techniques for editing objects.

Editing Text

Editing text requires that you target the input focus at the text to be edited. For mouse input, the input focus always coincides with the pointer (button down) location. For the pen, it is the point where the pen touches the screen. For the keyboard, the input focus is determined with the navigation keys. In all cases, the visual indication that a text field has the input focus is the presence of the text cursor, or insertion point.

Inserting Text

Inserting text involves the user placing the insertion point at the appropriate location and then typing. For each character typed, your application should move the insertion point one character to the right (or left, depending on the language).

If the text field supports multiple lines, *wordwrap* the text; that is, automatically move text to the next line as the textual input exceeds the width of the text-entry area.

Overtype Mode

Overtype is an optional text-entry behavior that operates similarly to the insertion style of text entry, except that you replace existing characters as new text is entered—with one character being replaced for each new character entered.

Use a block cursor that appears at the current character position to support overtype mode, as shown in Figure 5.3. This looks the same as the selection of that character and provides the user with a visual cue about the difference between the text-entry modes.

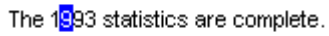


Figure 5.3 An overtype cursor

Use the INSERT key to toggle between the normal insert text-entry convention and overtype mode.

Deleting Text

The DELETE and BACKSPACE keys support deleting text. The DELETE key deletes the character to the right of the text insertion point. BACKSPACE removes the character to the left. In either case, move text in the direction of the deletion to fill the gap—this is sometimes referred to as *auto-joining*. Do not place deleted text on the Clipboard. For this reason, include at least a single-level undo operation in these contexts.

For a text selection, when the user presses DELETE or BACKSPACE, remove the entire block of selected text. Delete text selections when new text is entered directly or by a transfer command. In this case, replace the selected text by the incoming input.

For more information about transfer operations, see the "Transfer Operations" section later in this chapter.

Handles

Objects may include special control points, called *handles*; handles facilitate certain types of operations, such as moving, sizing, scaling, cropping, shaping, or auto-filling. The type of handle you use depends on the type of object. For example, for windows the title bar acts as a "move handle." The borders of the window act as "sizing handles." For icons, the selected icon acts as its own "move handle." In pen-enabled controls, special handles may appear for selection and access to the operations available for an object.

For more information about these handles, see the "Pen-Specific Editing Techniques" section later in this chapter.

A common form of handle is a square box placed at the edge of an object, as shown in Figure 5.4.

Figure 5.4 Handles

When the handle's interior is solid, the handle implies that it can perform a certain operation, such as sizing, cropping, or scaling. If the handle is "hollow," the handle does not currently support an operation. You can use such an appearance to indicate selection even when an operation is not available.

For more information about the design of handles, see Chapter 13, "Visual Design."

Transactions

A *transaction* is a unit of change to an object. The granularity of a transaction may span from the result of a single operation to that of a set of multiple operations. In an ideal model, transactions are applied immediately, and there is support for “rolling back,” or undoing, transactions. Because there are times when this is not practical, specific interface conventions have been established for committing transactions. If there are pending transactions in a window when it is closed, always prompt the user to ask whether to apply or discard the transactions.

Transactions can be committed at different levels, and a commitment made at one level may not imply a permanent change. For example, the user may change font properties of a selection of text, but these text changes may require saving the document file before the changes are permanent.

Use the following commands for committing transactions at the file level.

Command	Function
Save	Saves all interim edits, or checkpoints, to disk and begins a new editing session.
Close	Prompts the user to save any uncommitted edits. If confirmed, the interim edits are saved and the window is removed.

Note Use the Save command in contexts where committing file transactions applies to transactions for an entire file, such as a document, and are committed at one time. It may not necessarily apply for transactions committed on an individual basis, such as record-oriented processing.

On a level with finer granularity, you can use the following commands for common handling transactions within a file.

Command	Function
Repeat	Duplicates the last/latest user transaction.
Undo	Reverses the last, or specified, transaction.
Redo	Restores the most recent, or specified, "undone" transaction.
Apply	Commits any pending transactions, but does not remove the window.
Cancel	Discards any pending transactions and removes the window.

Following are the recommended commands for handling process transactions.

Command	Function
Pause	Suspends a process.
Resume	Resumes a suspended process.
Stop	Halts a process.

Note Although you can use the Cancel command to halt a process, Cancel implies that the state will be restored to what it was before the process was initiated.

Properties

Defining and organizing the properties of an application's components are a key part of evolving toward a more data-centered design. Commands such as Options, Info, Summary Info, and Format often describe features that can be redefined as properties of a particular object (or objects). The Properties command is the common command for accessing the properties of an object; when the user chooses this command, you typically display the property sheet for the selection.

For more information about property sheets, see Chapter 8, "Secondary Windows."
Defining how to provide access to properties for visible or easily identifiable objects, such as a selection of text, cells, or drawing objects, is straightforward. It may be more difficult to define how to access properties of less tangible objects, such as paragraphs. In some cases, you can include these properties by implication. For example, requesting the properties of a text selection can also provide access to the properties of the paragraph in which the text selection is included.

Another way to provide access to an object's properties is to create a representation of the object. For example, the properties of a page could be accessed through a graphic or other representation of the page in a special area (for example, the status bar) of the window.

Yet another technique to consider is to include specific property entries on the menu of a related object. For example, the pop-up menu of a text selection could include a menu entry for a paragraph. Or consider using the cascading submenu of the Properties command for individual menu entries, but only if the properties are not easily made accessible otherwise. Adding entries for individual properties can easily end up cluttering a menu.

The Properties command is not the exclusive means of providing access to the properties of an object. For example, folder views display certain properties of objects stored in the file system. In addition, you can use toolbar controls to display properties of selected objects.

Pen-Specific Editing Techniques

A pen is more than just a pointing device. When a standard pen device is installed, the system provides special interfaces and editing techniques.

Editing in Pen-Enabled Controls

If a pen is installed, the system automatically provides a special interface, called the *writing tool button*, to make text editing as easy as possible, enhance recognition accuracy, and streamline correction of errors. The writing tool interface, as shown in Figure 5.5, adds a button to your standard text controls. Because this effectively reduces the visible area of the text box, take this into consideration when designing their size.

Figure 5.5 A standard text box with writing tool button

Figure 5.6 shows how you can also add writing tool support for any special needs of your software.

Figure 5.6 Adding the writing tool button

When the text box control has the focus, a selection handle appears. The user can drag this handle to make a selection.

Tapping the writing tool button with a pen (or clicking it with a mouse) presents a special text editing window, as shown in Figure 5.7. Within this window, the user can write text that is recognized automatically.

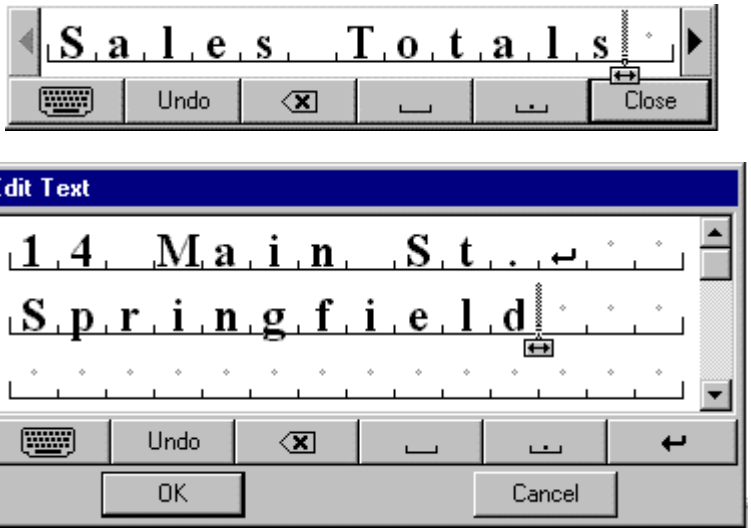


Figure 5.7 Single and multiline text editing windows

In the writing tool editing window, each character is displayed within a special cell. If the user selects text in the original text field, the writing tool window reflects that selection. The user can reset the selection to an insertion point by tapping between characters. This also displays a special selection handle that can be dragged to select multiple characters, as shown in Figure 5.8.

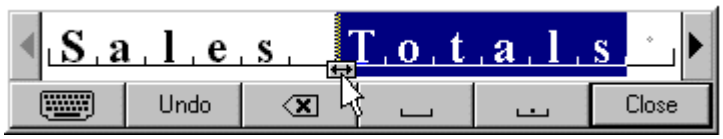


Figure 5.8 Selecting text with the selection handle

The user can select a single character in its cell by tapping, or double-tapping to select a word. When the user taps a single character, an action handle displays a list of alternative characters, as shown in Figure 5.9.

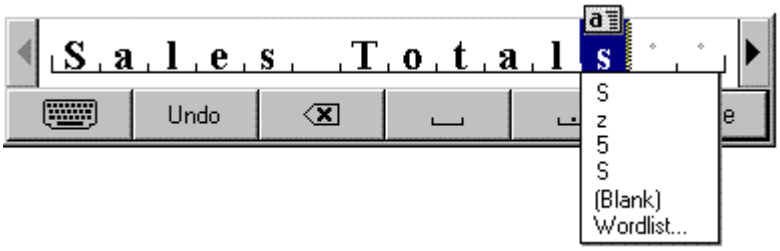


Figure 5.9 An action handle with a list of alternatives

Choosing an alternative replaces the selected character and removes the list. Writing over a character or tapping elsewhere also removes the list. The new character replaces the existing one and resets the selection to an insertion point placed to the left of the new character.

The list also includes an item labeled Wordlist. When the user selects this choice, the word that contains the character becomes selected and a list of alternatives is displayed, as shown in Figure 5.10. This list also appears when the user selects a complete word by double-tapping. Choosing an alternative replaces the selected word.

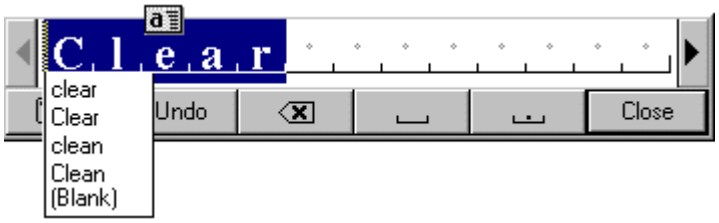


Figure 5.10 Tapping displays a list of Alternatives

When a selection exists in the window, an action handle appears; the user can use it to perform other operations on the selected items. For example, using the action handle moves or copies the selection by dragging, or the pop-up menu for the selection can be accessed by tapping on the handle, as shown in Figure 5.11.

For more information about pop-up menus, see Chapter 7, "Menus, Controls, and Toolbars."

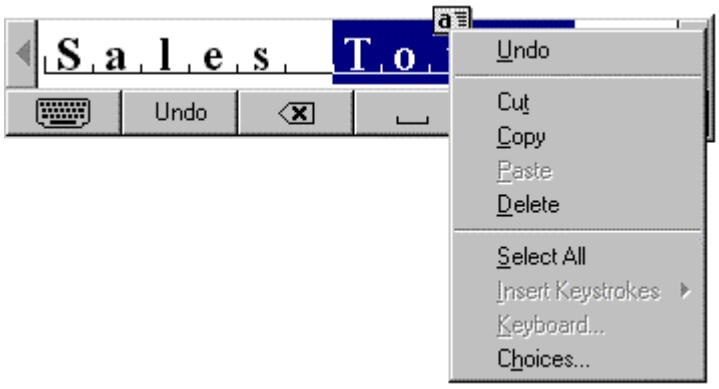


Figure 5.11 Tapping on the handle displays a pop-up menu

The buttons on the writing tool window provide for scrolling the text as well as common functions such as Undo, Backspace, Insert Space, Insert Period, and Close (for closing the text window). A multiline writing tool window includes Insert New Line.

The writing tool window also provides a button for access to an onscreen keyboard as an alternative to entering characters with the pen, as shown in Figure 5.12. The user taps the button with the corresponding keyboard glyph on it and the writing tool onscreen keyboard pop-up window replaces the normal writing tool window.

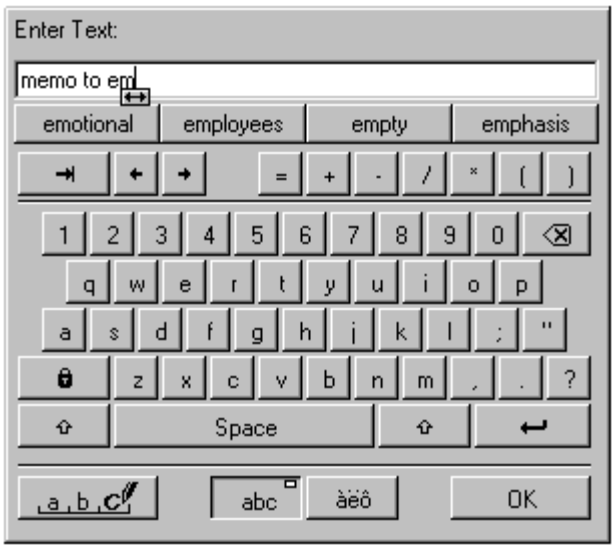


Figure 5.12 The writing toolpop-up window

The writing tool "remembers" its previous use—for text input or as an onscreen keyboard—and opens in the appropriate editing window when subsequently used. In addition, note that when the user displays a writing tool window, it gets the input focus, so avoid using the loss of input focus to a field as an indication that the user is finished with that field or that all text editing occurs directly within a text box.

Pen Editing Gestures

The pen, when used as a pointing device, supports editing techniques defined for the mouse; the pen also supports gestures for editing. Gestures (except for Undo) operate positionally, acting on the objects on which they are drawn. If the user draws a gesture on an unselected object, it applies to that object, even if a selection exists elsewhere within the same selection area. Any pending selections become unselected. If a user draws a gesture over both selected and unselected objects, however, it applies only to the selected ones. If a gesture is drawn over only one element of the selection, it applies to the entire selection. If the gesture is drawn in empty space (on the background), it applies to any existing selection within that selection scope. If no selection exists, the gesture has no effect.

For most gestures, the hot spot of the gesture determines specifically which object the gesture applies to. If the hot spot occurs on any part of a selection, it applies to the whole selection.

Table 5.2 lists the common pen editing gestures. For these gestures, the hot spot of the gesture is the area inside the circle stroke of the gesture.

Table 5.2 Basic Navigation Keys

Gesture	Name	Operation
	check-circle	Edit (displays the writing tool editing window) for text; Properties for all other objects.
	c-circle	Copy
	d-circle	Delete (or Clear)
	m-circle	Menu
	n-circle	New line
	p-circle	Paste
	s-circle	Insert space
	t-circle	Insert tab
	u-circle	Undo
	x-circle	Cut
	^-circle	Insert text

Note These gestures can be localized in certain international versions. In Kanji versions, the circle-k gesture is used to convert Kana to Kanji.

Transfer Operations

Transfer operations are operations that involve (or can be derived from) moving, copying, and linking objects from one location to another. For example, printing an object is a form of a transfer operation because it can be defined as copying an object to a printer.

Three components make up a transfer operation: the object to be transferred, the destination of the transfer, and the operation to be performed. You can define these components either explicitly or implicitly, depending on which interaction technique you use.

The operation defined by a transfer is determined by the destination. Because a transfer may have several possible interpretations, you can define a default operation and other optimal operations, based on information provided by the source of the transfer and the compatibility and capabilities of the destination. Attempting to transfer an object to a container can result in one of the following alternatives:

- Rejecting the object.
- Accepting the object.
- Accepting a subset or transformed form of the object (for example, extract its content or properties but discard its present containment, or convert the object into a new type).

Most transfers are based on one of the following three fundamental operations.

Operation	Description
Move	Relocates or repositions the selected object. Because it does not change the basic identity of an object, a move operation is not the same as copying an object and deleting the original.
Copy Makes a duplicate of an object. The resulting object is independent of its original. Duplication does not always produce an identical clone. Some of the properties of a duplicated object may be different from the original. For example, copying an object may result in a different name or creation date. Similarly, if some component of the object restricts copying, then only the unrestricted elements may be copied.	Makes a duplicate of an object. The resulting object is independent of its original. Duplication does not always produce an identical clone. Some of the properties of a duplicated object may be different from the original. For example, copying an object may result in a different name or creation date. Similarly, if some component of the object restricts copying, then only the unrestricted elements may be copied.

Link Creates a connection between two objects. The result is usually an object in the destination that provides access to the original.	Creates a connection between two objects. The result is usually an object in the destination that provides access to the original.

There are two different methods for supporting the basic transfer interface: the command method and the direct manipulation method.

Command Method

The command method for transferring objects uses the Cut, Copy, and Paste commands. Place these commands on the Edit drop-down menu as well as on the pop-up menu for a selected object. You can also include toolbar buttons to support these commands.

To transfer an object, the user:

1. Makes a selection.
2. Chooses either Cut or Copy.
3. Navigates to the destination (and sets the insertion location, if appropriate).
4. Chooses a Paste operation.

Cut removes the selection and transfers it (or a reference to it) to the Clipboard. Copy duplicates the selection (or a reference to it) and transfers it to the Clipboard. Paste completes the transfer operation. For example, when the user chooses Cut and Paste, remove the selection from the source and relocate it to the destination. For Copy and Paste, insert an independent duplicate of the selection and leave the original unaffected. When the user chooses Copy and Paste Link or Paste Shortcut, insert an object at the destination that is linked to the source.

Choose a form of Paste command that indicates how the object will be transferred into the destination. Use the Paste command by itself to indicate that the object will be transferred as native content. You can also use alternative forms of the Paste command for other possible transfers, using the following general form.

Paste [*short type name*] [**as** | **to** *object type* | *object name*]

For example, Paste Cells as Word Table, where [*short type name*] is Cells and [*object type*] is Word Table.

For more information about object names, including their short type name, see Chapter 10, "Integrating with the System."

The following summarizes common forms of the Paste command.

Command	Function
Paste	Inserts the object on the Clipboard as native content (data).
Paste [*short type name*]	Inserts the object on the Clipboard as an OLE embedded object. The OLE embedded object can be activated directly within the destination.
Paste [*short type name*] as Icon	Inserts the object on the Clipboard as an OLE embedded object. The OLE embedded object is displayed as an icon.
Paste Link	Inserts a data link to the object that was copied to the Clipboard. The object's value is integrated or transformed as native content within the destination, but remains linked to the original object so that changes to it are reflected in the destination.
Paste Link to [*object name*]	Inserts an OLE linked object, displayed as a picture of the object copied to the Clipboard. The representation is linked to the object copied to the Clipboard so that any changes to the original source object will be reflected in the destination.
Paste Shortcut	Inserts an OLE linked object, displayed as a shortcut icon, to the object that was copied to the Clipboard. The representation is linked to the object copied to the Clipboard so that any changes to the original source object will be reflected in the destination.
Paste Special	Displays a dialog box that gives the user explicit control over how to insert the object on the Clipboard.

For more information about these Paste command forms and the Paste Special dialog box, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Use the destination's context to determine what form(s) of the Paste operation to include based on what options it can offer to the user, which in turn may be dependent on the available forms of the object that its source location object provides. It can also be dependent on the nature or purpose of the destination. For example, a printer defines the context of transfers to it.

Typically, you will need only Paste and Paste Special commands. The Paste command can be dynamically modified to reflect the destination's default or preferred form by inserting the transferred object—for example, as native data or as an OLE embedded object. The Paste Special command can be used to handle any special forms of transfer. Although, if the destination's context makes it reasonable to provide fast access to another specialized form of transfer, such as Paste Link, you can also include that command.

Use the destination's context also to determine the appropriate side effects of the Paste operation. You may also need to consider the type of object being inserted by the Paste operation and the relationship of that type to the destination. The following are some common scenarios.

- When the user pastes into a destination that supports a specific insertion location, replace the selection in the destination with the transferred data. For example, in text or list contexts, where the selection represents a specific insertion location, replace the destination's active selection. In text contexts where there is an insertion location, but there is no existing selection, place the insertion point after the inserted object.
- For destinations with nonordered or Z-ordered contexts where there is no explicit insertion point, add the pasted object and make it the active selection. Use the destination's context also to determine where to place the pasted object. Consider any appropriate user contextual information. For example, if the user chooses the Paste command from a pop-up menu, you can use the pointer's location when the mouse button is clicked to place the incoming object. If the user supplies no contextual clues, place the object at the location that best fits the context of the destination—for example, at the next grid position.
- If the new object is automatically connected (linked) to the active selection (for example, table data and a graph), you may insert the new object in addition to the selection and make the inserted object the new selection.

You also use context to determine whether to display an OLE embedded or OLE linked object as content (view or picture of the object's internal data) or as an icon. For example, you can decide what presentation to display based on what Paste operation the user selects; Paste Shortcut implies pasting an OLE link as an icon. Similarly, the Paste Special command includes options that allow the user to specify how the transferred object should be displayed. If there is no user-supplied preference, the destination application defines the default. For documents, you typically display the inserted OLE object as in its content presentation. If icons better fit the context of your application, make the default Paste operation display the transferred OLE object as an icon.

The execution of a Paste command should not affect the content of the Clipboard. This allows data on the Clipboard to be pasted multiple times, although subsequent paste operations should always result in copies of the original. Remember that a subsequent Cut or Copy command will replace the last entry on the Clipboard.

Direct Manipulation Method

The command method is useful when a transfer operation requires the user to navigate between source and destination. However, for many transfers, direct manipulation is a natural and quick method. In a direct manipulation transfer, the user selects and drags an object to the desired location, but because this method requires motor skills that may be difficult for some users to master, avoid using it as the exclusive transfer method. The best interfaces support the transfer command method for basic operations as well as direct manipulation transfer as a shortcut.

When a pen is being used as a pointing device, or when it drags an action handle, it follows the same conventions as dragging with mouse button 1. For pens with barrel buttons, use the barrel+drag action as the equivalent of dragging with mouse button 2.

There is no keyboard interface for direct manipulation transfers.

You can support direct manipulation transfers to any visible object. The object (for example, a window or icon) need not be currently active. For example, the user can drop an object in an inactive window. This action activates the window. If an inactive object cannot accept a direct manipulation transfer, it (or its container) should provide feedback to the user.

How the transferred object is integrated and displayed in the drop destination is determined by the destination's context. A dropped object can be incorporated either as native data, as an OLE object, or as a partial form of the object such as its properties or a transformed object. You determine whether to add to or replace an existing selection based on the context of the operation, using such factors as the formats available for the object, the destination's purpose, and any user-supplied information such as the specific location that the user drops or commands (or modes) that the user has selected. For example, an application can supply a particular type of tool for copying the properties of objects.

Default Drag and Drop

Default drag and drop transfers an object using mouse button 1. How the operation is interpreted is determined by what the destination defines as the appropriate default operation. As with the command method, the destination determines this based on information about the object (and the formats available for the object) and the context of the destination itself. Avoid defining a destructive operation as the default. When that is unavoidable, display a message box to confirm the intentions of the user.

Using this transfer technique, the user can directly transfer objects between documents defined by your application as well as to system resources, such as folders and printers. Support drag and drop following the same conventions the system supports: the user presses button 1 down on an object, moves the mouse while holding the button down, and then releases the button at the destination. For the pen, the destination is determined by the location where the user lifts the pen tip from the screen.

The most common default transfer operation is Move, but the destination (dropped on object) can reinterpret the operation to be whatever is most appropriate. Therefore, you can define a default drag and drop operation to be another general transfer operation such as Copy or Link, a destination specific command such as Print or Send To, or even a specialized form of transfer such as Copy Properties.

Nondefault Drag and Drop

Nondefault drag and drop transfers an object using mouse button 2. In this case, rather than executing a default operation, the destination displays a pop-up menu when the user releases the mouse button, as shown in Figure 5.13. The pop-up menu contains the appropriate transfer completion commands.

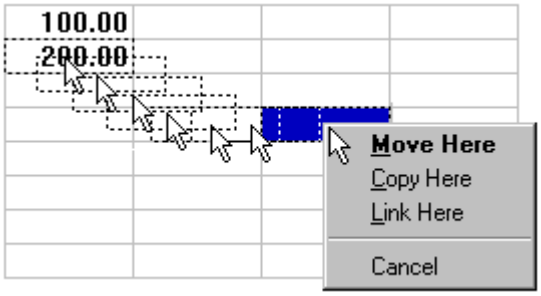


Figure 5.13 A nondefault drag and drop operation

The destination always determines which transfer completion commands to include on the resulting pop-up menu, usually factoring in information about the object supplied by the source location.

The form for nondefault drag and drop transfer completion verbs follows similar conventions as the Paste command. Use the common transfer completion verbs, Move Here, Copy Here, and Link Here, when the object being transferred is native data of the destination. When it is not, include the short type name. You can also display alternative completion verbs that communicate the context of the destination; for example, a printer displays a Print Here command. For commands that support only a partial aspect or a transformation of an object, use more descriptive indicators—for example, Copy Properties Here, or Transpose Here.

Use the following general form for nondefault drag and drop transfer commands.

[*Command Name*] [*object type* | *object name*] **Here** [**as** *object type*]

The following summarizes command forms for nondefault transfer completion commands.

Command	**Function**
Move Here	Moves the selected object to the destination as native content (data).
Copy Here	Creates a copy of the selected object in the destination as native content.
Link Here	Creates a data link between the selected object and the destination. The original object's value is integrated or transformed as native data within the destination, but remains linked to the original object so that changes to it are reflected in the destination.
Move [*short type name*] Here Copy [*short type name*] Here	Moves or copies the selected object as an OLE embedded object. The OLE embedded object is displayed in its content presentation and can be activated directly within the destination.
Link [*short type name*] Here	Creates an OLE linked object displayed as a picture of the selected object. The representation is linked to the selected object so that any changes to the original object will be reflected in the destination.
Move [*short type name*] Here as Icon Copy [*short type name*] Here as Icon	Moves or copies the selected object as an OLE embedded object and displays it as an icon.
Create Shortcut Here	Creates an OLE linked object to the selected object; displayed as a shortcut icon. The representation is linked to the selected object so that any changes to the original object will be reflected in the destination.

Define and appropriately display one of the commands in the pop-up menu to be the default drag and drop command. This is the command that corresponds to the effect of dragging and dropping with mouse button 1.

For more information about how to display default menu commands, see Chapter 13, "Visual Design."

Canceling a Drag and Drop Transfer

When a user drags and drops an object back on itself, interpret the action as cancellation of a direct manipulation transfer. Similarly, cancel the transfer if the user presses the ESC key during a drag transfer. In addition, include a Cancel command in the pop-up menu of a nondefault drag and drop action. When the user chooses this command, cancel the operation.

Differentiating Transfer and Selection When Dragging

Because dragging performs both selection and transfer operations, provide a convention that allows the user to differentiate between these operations. The convention you use depends on what is most appropriate in the current context of the object, or you can provide specialized handles for selection or transfer. The most common technique uses the location of the pointer at the beginning of the drag operation. If the pointer is within an existing selection, interpret the drag to be a transfer operation. If the drag begins outside of an existing selection, on the background's white space, interpret the drag as a selection operation.

Scrolling When Transferring by Dragging

When the user drags and drops an object from one scrollable area (such as a window, pane, or list box) to another, some tasks may require transferring the object outside the boundary of the area. Other tasks may involve dragging the object to a location not currently in view. In this latter case, it is convenient to automatically scroll the area (also known as *automatic scrolling* or autoscroll) when the user drags the object to the edge of that scrollable area. You can accommodate both these behaviors by using the velocity of the dragging action. For example, if the user is dragging the object slowly at the edge of the scrollable area, you scroll the area; if the object is being dragged quickly, do not scroll.

To support this technique, during a drag operation you sample the pointer's position at the beginning of the drag, each time the mouse moves, and on an application-set timeout (every 100 milliseconds recommended). Store each value in an array large enough to hold at least three samples, replacing existing samples with later ones. Then calculate the pointer's velocity based on at least the last two locations of the pointer.

To calculate the velocity, sum the distance between the points in each adjacent sample and divide the total by the sum of the time elapsed between samples. Distance is the absolute value of the difference between the x and y locations, or (abs(x1 − x2) + abs(y1 − y2)). Multiply this by 1024 and divide it by the elapsed time to produce the velocity. The 1024 multiplier prevents the loss of accuracy caused by integer division.

> **Note** Distance as implemented in this algorithm is not true Cartesian distance. This implementation uses an approximation for purposes of efficiency, rather than using the square root of the sum of the squares, (sqrt((x1 – x2)^2 + (y1 – y2)^2)), which is more computationally expensive.

You also predefine a hot zone along the edges of the scrollable area and a scroll timeout value. The recommended hot zone width is XXX. You use the scroll timeout value to control the scroll rate. During the drag operation, scroll the area if the following conditions are met: the user moves the pointer within the hot zone, the current velocity is below a certain threshold velocity, the scroll timeout has elapsed, and the scrollable area is able to scroll in the direction associated with the hot zone it is in. The recommended threshold velocity is XXX. These principles are illustrated in Figure 5.14.

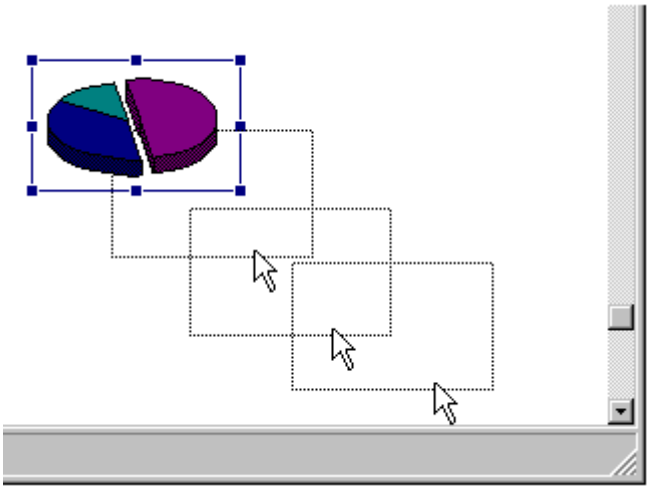


Figure 5.14 Automatic scrolling based on velocity of dragging

> **Note** The system provides support for automatic scrolling using the **AddTimeSample** function and the **AUTO_SCROLL_DATA** structure to record and maintain time and position samples and the scroll timeout. For more information about using these API elements, see the *Microsoft Win32 Programmer's Reference*.

Transfer Feedback

Because transferring objects is one of the most common user tasks, providing appropriate feedback is an important design factor. Inconsistent or insufficient feedback can result in user confusion.

For more information about the design of transfer feedback, see Chapter 13, "Visual Design."

Command Method Transfers

For a command method transfer, remove the selected object visually when the user chooses the Cut command. If there are special circumstances that make removing the object's appearance impractical, you can instead display the selected object with a special appearance to inform the user that the Cut command was completed, but that the object's transfer is pending. For example, the system displays icons in a checkerboard dither to indicate this state. But the user will expect Cut to remove a selected object, so carefully consider the impact of inconsistency if you choose this alternate feedback.

The Copy command requires no special feedback. A Paste operation also requires no further feedback than that already provided by the insertion of the transferred object. However, if you did not remove the display of the object and used an alternate representation when the user chose the Cut command, you must remove it now.

Direct Manipulation Transfers

During a direct manipulation transfer operation, provide visual feedback for the object, the pointer, and the destination. Specifically:

- Display the object with selected appearance while the view it appears in has the focus. To indicate that the object is in a transfer state, you can optionally display the object with some additional appearance characteristics. For example, for a move operation, use the checkerboard dithered appearance used by the system to indicate when an icon is Cut. Change this visual state based on the default completion operation supported by the destination the pointer is currently over. Retain the representation of the object at the original location until the user completes the transfer operation. This not only provides a visual cue to the nature of the transfer operation, it provides a convenient visual reference point.
- Display a representation of the object that moves with the pointer. Use a presentation that provides the user with information about how the information will appear in the destination and that does not obscure the context of the insertion location. For example, when transferring an object into a text context, it is important that the insertion point not be obscured during the drag operation. A translucent or outline representation, as shown in Figure 5.15, works well because it allows the underlying insertion location to be seen while also providing information about the size, position, and nature of the object being dragged.

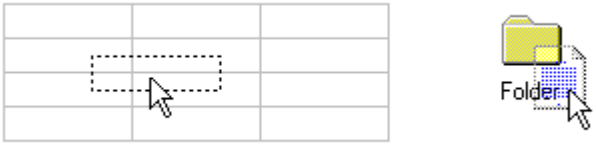


Figure 5.15 Outline and translucent representations for transfer operations

- The object's existing source location provides the transferred object's initial appearance, but any destination can change the appearance. Design the presentation of the object to provide feedback as to how the object will be integrated by that destination. For example, if an object will be embedded as an icon, display the object as an icon. If the object will be incorporated as part of the native content of the destination, then the presentation of the object that the destination displays should reflect that. For example, if a table being dragged into a document will be incorporated as a table, the representation could be an outline or translucent form of the table. On the other hand, if the table will be converted to text, display the table as a representation of text, such as a translucent presentation of the first few words in the table.

- Display the pointer appropriate to the context of the destination, usually used for inserting objects. For example, when dragging an object into a text editing context such that the object will be inserted between characters, display the usual text editing pointer (sometimes called the I-beam pointer).
- Display the interpretation of the transfer operation at the lower right corner of the pointer, as shown in Figure 5.16. No additional glyph is required for a move operation. Use a plus sign (+) when the transfer is a copy operation. Use the shortcut arrow graphic for linking.

Figure 5.16 Pointers – move, copy, and link operations

- Use visual feedback to indicate the receptivity of potential destinations. You can use selection highlighting and optionally animate or display a representation of the transfer object in the destination. Optionally, you can also indicate when a destination cannot accept an object by using the "no drop" pointer when the pointer is over it, as shown in Figure 5.17.

Figure 5.17 A "no drop " pointer

Specialized Transfer Commands

In some contexts, a particular form of a transfer operation may be so common, that introducing an additional specialized command is appropriate. For example, if copying existing objects is a frequent operation, you can include a Duplicate command. Following are some common specialized transfer commands.

Command	Function
Delete	Removes an object from its container. If the object is a file, the object is transferred to the Recycle Bin.
Clear	Removes the content of a container.
Duplicate	Copies the selected object.
Print	Prints the selected object on the default printer.
Send To	Displays a list of possible transfer destinations and transfers the selected object to the user selected destination.

Note Delete and Clear are often used synonymously. However, they are best differentiated by applying Delete to an object and Clear to the container of an object.

Shortcut Keys for Transfer Operations

Following are the defined shortcut techniques for transfer operations.

Shortcut	Operation
CTRL+X	Performs a Cut command.
CTRL+C	Performs a Copy command.
CTRL+V	Performs a Paste command.
CTRL+drag	Toggles the meaning of the default direct manipulation transfer operation to be a copy operation (provided the destination can support the copy operation). The modifier may be used with either mouse button.
ESC	Cancels a drag and drop transfer operation.

Because of the wide use of these command shortcut keys throughout the interface, do not reassign them to other commands.

For more information about reserved and recommended shortcut key assignments, see Appendix B, "Keyboard Interface Summary."

Scraps

The system allows the user to transfer objects within a data file to the desktop or folders providing that the application supports the OLE transfer protocol. The result of the transfer operation is a file icon called a *scrap*. When the user transfers a scrap into an application, integrate it as if it were being transferred from its original source. For example, if a selected range of cells from a spreadsheet is transferred to the desktop, they become a scrap. If the user transfers the resulting scrap into a word processing document, incorporate the cells as if they were transferred directly from the spreadsheet. Similarly, if the user transfers the scrap back into the spreadsheet, integrate the cells as if they were originally transferred within that spreadsheet.

Creation Operations

Creating new objects is a common user action in the interface. Although applications can provide the context for object creation, avoid considering an application's interface as the exclusive means of creating new objects. Creation is typically based on some predefined object or specification and can be supported in the interface in a number of ways.

Copy Command

Making a copy of an existing object is the fundamental paradigm for creating new objects. Copied objects can be modified and serve as prototypes for the creation of other new objects. The transfer model conventions define the basic interaction techniques for copying objects. Copy and Paste commands and drag and drop manipulation provide this interface.

New Command

The New command facilitates the creation of new objects. New is a command applied to a specific object, automatically creating a new instance of the object's type. The New command differs from the Copy and Paste commands in that it is a single command that generates a new object.

Insert Command

The Insert command works similarly to the New command, except that it is applied to a container to create a new object, usually of a specified type, in that container. In addition to inserting native types of data, use the Insert command to insert objects of different types. By supporting OLE, you can support the creation of a wide range of objects. In addition, objects supported by your application can be inserted into data files created by other OLE applications.

For more information about inserting objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Using Controls

You can use controls to support the automatic creation of new objects. For example, in a drawing application, buttons are often used to specify tools or modes for the creation of new objects, such as drawing particular shapes or controls. Buttons can also be used to insert OLE objects.

For more information about using buttons to create new objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects." For general information about using controls to perform operations, see Chapter 7, "Menu, Controls, and Toolbars."

Using Templates

A *template* is an object that automates the creation of a new object. To distinguish its purpose, display a template icon as a pad with the small icon of the type of the object to be created, as shown in Figure 5.18.

Figure 5.18 A template icon

Define the New command as the default operation for a template object; this starts the creation process, which may either be automatic or request specific input from the user. Place the newly created object in the same location as the container of the template. If circumstances make that impractical, place the object in a common location such as the desktop, or, during the creation process, include a prompt that allows a user to specify some other destination. In the former situation, display a message box so that the user knows where the object will appear.

Operations on Linked Objects

A *link* is a connection between two objects that represents or provides access to another object that is in another location in the same container or in a different, separate container. The components of this relationship include the link source (sometimes referred to as the referent) and the link or linked object (sometimes referred to as the reference). A linked object often has operations and properties independent of of its source. For example, a linked object's properties can include attributes like update frequency, the path description of its link source, and the appearance of the linked object. The containers in which they reside provide access to and presentation of commands and properties of linked objects.

Links can be presented in various ways in the interface. For example, a *data link* propagates a value between two objects, such as between two cells in a worksheet or a series of data in a table and a chart. *Jumps* (also referred to as hyperlinks) provide navigational access to another object. An *OLE linked object* provides access to any operation available for its link source and also supplies a presentation of the link source. A shortcut icon is a link, displayed as an icon.

For more information about OLE linked objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects"

When the user transfers a linked object, store both the absolute and relative path to its link source. The absolute path is the precise description of its location, beginning at the root of its hierarchy. The relative path is the description of its location relative to its current container.

The destination of a transfer determines whether to use the absolute or relative path when the user accesses the link source through the linked object. The relative path is the most common default path. However, regardless of which path you use, if it fails, use the alternative path. For example, if the user copies a linked object and its link source to another location, the result is a duplicate of the linked object and the link source. The relative path for the duplicate linked object is the location of the duplicate of the link source. The absolute path for the duplicate linked object is the description of the location of the initial link source. Therefore, when the user accesses the duplicate of the linked object, its inferred connection should be with the duplicate of the link source. If that connection fails – for example, because the user deletes the duplicate of the linked source – use the absolute path, the connection to the original link source.

You can optionally make the preferred path for a linked object a field in the property sheet for linked object. This allows the user to choose whether to have a linked object make use of the absolute or relative path to its link source.

When the user applies a link operation to a linked object, link to the linked object rather than its linked source. That is, linking a linked object results in a linked object linked to a linked object. If such an operation is not valid or appropriate – for example, because the linked object provides no meaningful context – then disable any link commands or options when the user selects a linked object.

Activation of a linked object depends on the kind of link. For example, a single click can activate a jump; however, it only results in selecting a data link or an OLE linked object. If you use a single click to do anything other than select the linked object, distinguish the object by either presenting it as a button control, displaying the hand pointer (as shown in Figure 5.19) when the user moves the pointer over the linked object, or both.

Figure 5.19 A hand pointer

CHAPTER 6

Windows

Windows provide the fundamental way a user views and interacts with data. Consistency in window design is particularly important because it enables users to easily transfer their learning skills and focus on their tasks rather than learn new conventions. This chapter describes the common window types and presents guidelines for general appearance and operation.

Common Types of Windows

Because windows provide access to different types of information, they can be classified according to common usage. Interacting with objects typically involves a *primary window* in which most primary viewing and editing activity takes place. In addition, multiple supplemental *secondary windows* can be included to allow users to specify parameters or options, or to provide more specific details about the objects or actions included in the primary window.

For more information about secondary windows, see Chapter 8, "Secondary Windows."

Primary Window Components

A typical primary window consists of a frame (or border) which defines its extent, and a title bar which identifies what is being viewed in the window. If the viewable content of the window exceeds the current size of the window, scroll bars are used. The window can also include other components like menu bars, toolbars, and status bars.

For more information about these components, see Chapter 7, "Menus, Controls, and Toolbars," and the section, "Scrolling Windows," later in this chapter.

Figure 6.1 shows the common components of a primary window.

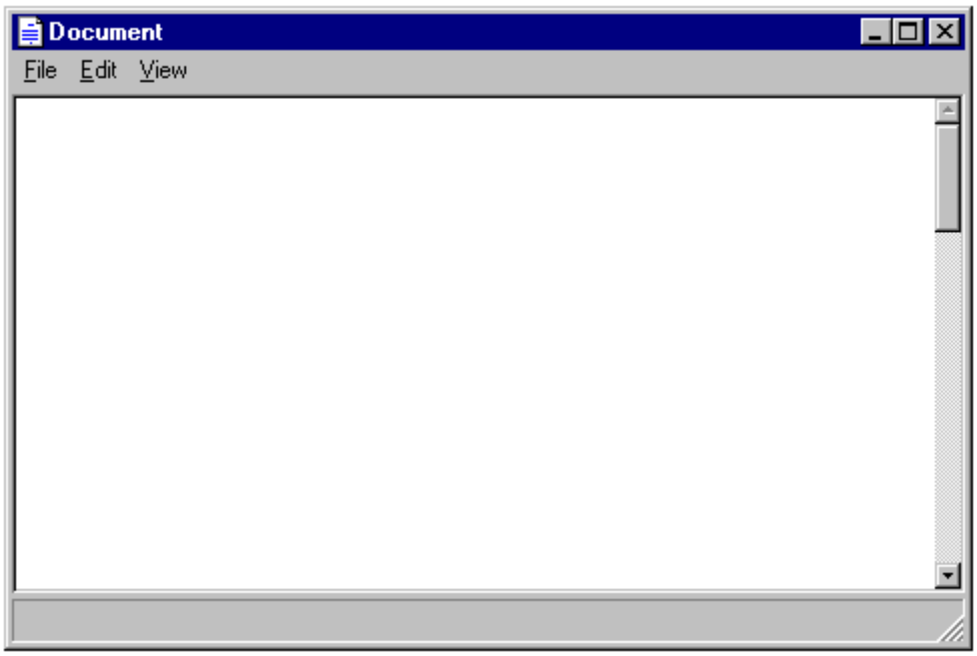


Figure 6.1 A primary window

Window Frames

Every window has a boundary that defines its shape. A sizable window has a distinct border that provides central points (handles) for resizing the window using direct manipulation. If the window cannot be resized, the border coincides with the edge of the window.

Title Bars

At the top edge of the window, inside its border, is the *title bar* (also referred to as the caption or caption bar), which extends across the width of the window. The title bar identifies what the window is viewing. It also serves as a control point for moving the window and an access point for commands that apply to the window and its associated view. For example, clicking on the title bar with mouse button 2 displays the pop-up menu for the window. Pressing the ALT+SPACEBAR key combination also displays the pop-up menu for the window.

For more information about pop-up menus, see Chapter 7, "Menus, Controls, and Toolbars."

Title Bar Icons

A small version of the object's icon appears in the upper left corner of the title bar; it represents the object being viewed in the window. If the window represents a "tool" application (that is, an application that does not create, load, and save separate data files), insert the small version of the application's icon in its title bar, as shown in Figure 6.2.

Figure 6.2 "Tool" title bar

If the application loads and saves documents or data files, place the icon that represents its document or data file type in the title bar, as shown in Figure 6.3.

Figure 6.3 Document title bar

For information about how to register icons for your application and data file types, see Chapter 10, "Integrating with the System." For more information about designing icons, see Chapter 13, "Visual Design."

If an application uses the multiple document interface (MDI), place the application's icon in the parent window's title bar, and place an icon that reflects the application data file type in the child window's title bar, as shown in Figure 6.4.

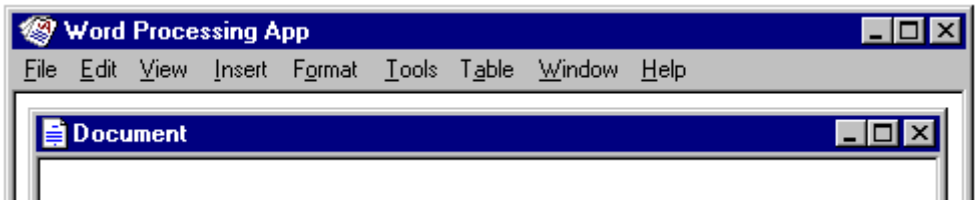


Figure 6.4 MDI application and document title bars

However, when a user maximizes the child window, hide the title bar, and merge its title information with the parent, as shown in Figure 6.5. Then display the icon from the child window's title bar in the menu bar of the parent window. If multiple child windows are open within the MDI parent window, display the icon from the active (topmost) child window.

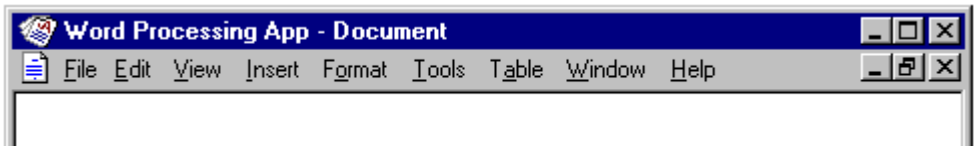


Figure 6.5 MDI parent window title bar with maximized child window

For more information about MDI, see Chapter 9, "Window Management."

When the user clicks the title bar icon with mouse button 2, display the pop-up menu for the object. Typically, the menu contains the same set of commands available for the icon from which the window was opened, except that Close replaces Open. Close is also the default command, so when the user double-clicks the title bar icon, the window closes.

Note When the user clicks the title bar icon with mouse button 1, the pop-up menu for the window is displayed. However, this behavior is only supported for backward compatibility with Windows 3.1. Avoid documenting it as the primary way to access the pop-up menu for the window.

Title Text

The title text is a label that identifies the name of the object being viewed through the window. It should correspond to the current icon in the title bar. For example, if a document or data file is displayed in the window, display the name of the file. It is also optional to include the name of the application in use; however, if it is used, the name of the file appears first, followed by a dash and the application name, as shown in Figure 6.6.

Figure 6.6 Title text order: document name — application name

Note The order of the document (or data) filename and application name differs from the Windows 3.1 guidelines. The new convention is better suited for the design of a data-centered interface.

If the window represents a "tool" application without any associated data files, such as the Windows Calculator, display the application's name in the title bar. If the tool application includes a specifier then include a dash and the specification text. For example, the Windows Find File application indicates a specification of search criteria in the title bar. Similarly, the Windows Explorer indicates what container the user is exploring. While this may appear to be inconsistent with the guideline for data files, the key distinction is what icon you display in the title bar. If that icon represents the application, then display the application name first. If the icon represents the data file, display the data filename first.

For an MDI application, use the application's name in the parent window and the data file's name in the child windows. When the user maximizes the file's child window, format the title text following the same convention as a tool application, with the application's name first, followed by the data filename..

If a data file currently has no user-supplied name, create one automatically by using the short type name—for example Document *n*, Sheet *n*, Chart *n*, where *n* is a number (as in Document 1). Use this name in the title text and also as the proposed default filename for the object in the Save As dialog box. If it is impractical or inappropriate to supply a default name, display a placeholder in the title, such as (Untitled).

For more information about short type names, see Chapter 10, "Integrating with the System." For more information about the Save dialog box, see Chapter 8, "Secondary Windows."

Display the text exactly as it appears to the user in the file system, using both uppercase and lowercase letters. Avoid including the file extensions or the path name in the title bar. This information is not meaningful for most users and can make it more difficult for them to identify the file. However, the system does provide an option for users to display filename extensions. Support this option by using the **SHGetFileInfo** function to format and display the filename appropriately based on the user's preference.

For more information about **SHGetFileInfo**, see the *Microsoft Win32 Programmer's Reference*.

If the name of the displayed object in the window changes—for example, when the user edits the name in the object's property sheet—update the title text to reflect that change. Always maintain a clear association between the object and its open window.

The title text and icon always represent the outmost container — the object that was opened—even if the user selects an embedded object or navigates the internal hierarchy of the object being viewed in the window. If you need an additional specifier to clarify what the user is viewing, place this specifier after the filename and clearly separated from the filename, such as enclosed in parentheses—for example, My HardDisk (C:). Because the system now supports long filenames, avoid additional specification whenever possible. Complex or verbose additions to the title text also make it more difficult for the user to easily read and identify the window.

When the width of the window does not allow you to display the complete title text, you may abbreviate the title text, being careful to maintain the essential information that allows the user to quickly identify the window.

For more information about abbreviating names, see Chapter 10, "Integrating with the System."

Avoid drawing directly into the title area or adding other controls. Such added items can make reading the name in the title difficult, particularly because the size of the title bar varies with the size of the window. In addition, the system uses this area for displaying special controls. For example, in some international versions of Windows, the title area provides information or controls associated with the input of certain languages.

Title Bar Buttons

Command buttons associated with the commands of the window appear in the title bar. They act as shortcuts to specific window commands. Clicking a title bar button with mouse button 1 invokes the command associated with the command button. When the user clicks a command button with mouse button 2, display the pop-up menu for the window. For the pen, tapping a window button invokes its associated command, and barrel-tapping it (or using the pen menu gesture) displays the pop-up menu for the window.

Typically, the following buttons appear in a primary window (provided that the window supports the respective functions).

Command button	Operation
	Closes the window.
	Minimizes the window.
	Maximizes the window.
	Restores the window.

When the commands are supported, by a window do not display the buttons.

Basic Window Operations

The basic operations for a window include: activation and deactivation, opening and closing, moving and sizing, and scrolling and splitting. The following sections describe these operations.

Activating and Deactivating Windows

While the system supports the display of multiple windows, the user generally works within a single window at a time. This window is called the *active window*. The active window is typically at the top of the window Z order. It is also visually distinguished by its title bar that is displayed in the active window title color (also referred to as the COLOR_ACTIVECAPTION value). All other windows are *inactive* with respect to the user's input; that is, while other windows can have ongoing processes, only the active window receives the user's input. The title bar of an

inactive window displays the system inactive window color of the system (the COLOR_INACTIVECAPTION value).

For more information about the COLOR_ACTIVECAPTION and COLOR_INACTIVECAPTION values, and the **GetSysColor** function, see the *Microsoft Win32 Programmer's Reference*.

The user activates a primary window by switching to it; this inactivates any other windows. To activate it with the mouse or pen, the user clicks or taps on any part of the window, including its interior. If the window is minimized, the user clicks (taps) the button representing the window in the taskbar. From the keyboard, the system provides the ALT+TAB key combination for switching between primary windows. (The SHIFT+ALT+TAB key also switches between windows, but in reverse order.) The reactivation of a window does not affect any pre-existing selection; the selection and focus are restored to the previously active state.

When the user reactivates a primary window, the window and all its secondary windows come to the top of the window order and maintain their relative positions. If the user activates a secondary window, its primary window comes to the top of the window order along with the primary window's other secondary windows.

When a window becomes inactive, hide the selection feedback (for example, display of highlighting or handles) of any selection within it to prevent confusion over which window is receiving keyboard input. A direct manipulation transfer (drag and drop) is an exception. Here, you can display transfer feedback if the pointer is over the window during the drag operation. Do not activate the window unless the user releases the mouse button (pen tip is lifted) in that window.

For more information about selection and transfer appearance, see Chapter 13, "Visual Design."

Opening and Closing Windows

When the user opens a primary window, include an entry for it on the taskbar. If the window has been opened previously, restore the window to its size and position when it was last closed. If possible and appropriate, reinstate the other related view information, such as selection state, scroll position, and type of view. When opening a primary window for the first time, open it to a reasonable default size and position as best defined by the object or application.

Opening the primary window activates that window and places it at the top of the window order. If the user attempts to open a primary window that is already open within the same desktop, follow these recommendations:

File type	Action when repeating an open operation
Document or data file	Activates the existing window of the object and displays it at the top of the window Z order.
Application file	Displays a message box indicating that an open window of that application already exists and offers the user the option to switch to the open window or to open another window. Either choice activates the selected window and brings it to the top of the window Z order.
Document file that is already open in a multiple document interface (MDI) application window	Activates the existing window of the file. Its MDI parent window comes to the top of the window Z order, and the file appears at the top of the Z order within its MDI parent window.
Document file that is not already open, but its associated multiple document interface (MDI) application is already running (open)	Opens a new instance of the file's associated MDI application at the top of the window Z order and displays the child window for the file.

For more information about MDI, see Chapter 9, "Window Management."

When opening a window, consider the size and orientation of the current screen upon which it will be opened. For example, on some systems, the display may be *landscape* oriented (long dimension along the bottom) and on others it may be *portrait*. The display resolution may vary as well. In such cases, adjust the size and position of the window from its stored state so that it appears relative to and yet completely on the user's display configuration.

The user closes a primary window by clicking (for a pen, tapping the screen) the Close button in the title bar or choosing the Close command from the window's pop-up menu. In addition, you can support double-clicking (with a pen, double-tapping the screen) on the title bar icon as a shortcut for closing the window for compatibility with previous versions of Windows.

When the user chooses the Close command or any other command that results in closing the primary window (for example, Exit or Shut Down), display a message asking the user whether to save any changes, discard any changes, or cancel the Close operation before closing the window. This gives the user control over any pending transactions that are not automatically saved. If there are no pending transactions, close the window.

For more information about supporting the Close command, see Chapter 5, "General Interaction Techniques."

When closing the primary window, close any of its dependent secondary windows as well. The design of your application affects whether closing the primary window also ends the application processes. For example, closing the window of a text document typically halts any application code or processes remaining for inputting or formatting text. However, closing the window of a printer has no effect on the jobs in the printer's queue. In both cases, closing the window removes its entry from the taskbar.

Moving Windows

The user can move a window either by dragging its title bar using the mouse or pen or by using the Move command on the window's pop-up menu. On most configurations, an outline representation moves with the pointer during the operation, and the window is redisplayed in the new location after the completion of the move. (The system also provides a display property setting that redraws the window dynamically as it is moved.) After choosing the Move command, the user can move the window with the keyboard interface by using arrow keys and pressing the ENTER key to end the operation and establish the window's new location. Never allow the user to reposition a window such that it cannot be accessed.

A window need not be active before the user can move it. The implicit action of moving the window activates it.

Moving a window can clip or reveal information shown in the window. In addition, activation can affect the view state of the window—for example, the current selection can be displayed. However, when the user moves a window, avoid making any changes to the content being viewed in that window.

Resizing Windows

Make your primary windows resizable unless the information displayed in the window is fixed, such as in the Windows Calculator program. The system provides several conventions that support user resizing of a window.

Sizing Borders

The user resizes a primary window by dragging the sizing border with the mouse or pen at the edge of a window or by using the Size command on the window's menu. On most configurations, an outline representation of the window moves with the pointer. (A display property setting allows the user to have the system dynamically redraw the window as it is sized.) After completing the size operation, the window assumes its new size. Using the keyboard, the user can size the window by choosing the Size command, using the arrow keys, and pressing the ENTER key.

A window does not need to be active before the user can resize it. The action of sizing the window implicitly makes it active, and it remains active after the sizing operation.

When the user resizes a window to be smaller, you must determine how to display the information being viewed in that window. Use the context and type of information to help you choose your approach. The most common

approach is to clip the information. However, in other situations where you want the user to see as much information as possible, you may want to consider using different methods, such as rewrapping or scaling the information. Use these variations carefully because they may not be consistent with the resizing behavior of most windows. In addition, avoid these methods when readability or maintaining the structural relationship of the information is important.

While the size of a primary window may vary, based on the user's preference, you can define a window's maximum and minimum size. When defining these sizes, consider the reasonable usage within the window, and the size and orientation of the screen.

Maximizing Windows

Although the user may be able to directly resize a window to its maximum size, the Maximize command optimizes this operation. The command is available on a window's pop-up menu, and as the Maximize command button in the title bar of a window.

Maximizing a window increases the size of the window to its largest, optimum size. The system default setting for the maximum size is as large as the display, excluding the space used by the taskbar. For an MDI child window, the default maximize size fills the parent window. However, you can define the size to be less (or, in some cases, more) than the display dimensions. Because display resolution and orientation varies, your software should not assume a fixed display size, but rather adapt to the shape and size defined by the system. If you use a standard system interface, such as the **SetWindowPlacement** function, the system automatically places your windows relative to the current display configuration.

For more information about the **SetWindowPlacement** function, see the *Microsoft Win32 Programmer's Reference*.

When the user maximizes a window, replace the Maximize button with a Restore button. In addition, disable the Maximize command and enable the Restore command on the pop-up menu for the window.

Minimizing Windows

Minimizing a window reduces it to its smallest size. To minimize a window, the user chooses the Minimize command on the window's pop-up menu or the Minimize command button on the title bar. For primary windows, minimizing removes the window from the screen, but leaves its entry in the taskbar. For MDI child windows, the window resizes to a minimum size within its parent window.

Note The Windows 3.1 representation of a minimized window using an icon is no longer appropriate. To reflect some status information about the open, but minimized, window, place the entry on the taskbar. For more information about status notification, see Chapter 10, "Integrating with the System."

When the user minimizes a window, disable the Minimize command on the pop-up menu for the window and enable the Restore command.

Restoring Windows

After maximizing or minimizing a window, the user can restore it to its previous size and position using the Restore command. For maximized windows, make this command available from the window's pop-up menu or the button which replaces the Maximize button in the title bar of the window.

For minimized, primary windows, enable the Minimize command in the pop-up menu of the window. The user restores a minimized primary window to its former size and position by clicking (for pens, tapping the screen) on its button in the taskbar that represents the window, selecting the Restore command on the pop-up menu of the window's taskbar button, or using the ALT+TAB key combination or the SHIFT+ALT+TAB key combination.

Size Grip

When you define a sizable window, include a size grip. A *size grip* is a special handle for sizing a window. It is not exclusive to the sizing border. To size the window, the user drags the grip and the window resizes following the same conventions as the sizing border.

Always locate the size grip in the lower right corner of the window. Typically, this means you place the size grip at the junction of a horizontal or vertical scroll bar, or at the right end of a horizontal scroll bar, or the bottom of a vertical scroll bar. However, if you include a status bar in the window, display the size grip at the far corner of the status bar instead. Never display the size grip in both locations at the same time.

For more information on the use of the size grip in a status bar, see Chapter 7, "Menus, Controls, and Toolbars."

Scrolling Windows

When the information viewed in a window exceeds the size of that window, the window should support scrolling. Scrolling enables the user to view portions of the object that are not currently visible in a window. Scrolling is commonly supported through the use of a scroll bar. A *scroll bar* is a rectangular control consisting of *scroll arrows*, a *scroll box*, and a *scroll bar shaft*, as shown in Figure 6.7.

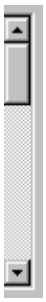


Figure 6.7 Scroll bar and its components

You can include a vertical scroll bar, a horizontal scroll bar, or both. The scroll bar aligns with the vertical or horizontal edge of the window orientation it supports. If the content is never scrollable in a particular direction, do not include a scroll bar for that direction.

Scroll bars are also available as separate window components. For more information about scroll bar controls, see Chapter 7, "Menus, Controls, and Toolbars."

The common practice is to display scroll bars if the view requires some scrolling under any circumstances. If the window becomes inactive or resized so that its content does not require scrolling, continue to display the scroll bars. While removing the scroll bars potentially allows the display of more information as well as feedback about the state of the window, it also requires the user to explicitly activate the window to scroll. Consistently displaying scroll bars also provides a more stable environment.

Scroll Arrows

Scroll arrow buttons appear at each end of a scroll bar, pointing in opposite directions away from the center of the scroll bar. The scroll arrows point in the direction that the window "moves" over the data. When the user clicks (for pens, tapping the screen) a scroll arrow, the data in the window moves, revealing information in the direction of the arrow in appropriate increments. The granularity of the increment depends on the nature of the content and context, but it is typically based on the size of a standard element. For example, you can use one line of text for vertical scrolling, one row for spreadsheets. You can also use an increment based a fixed unit of measure. Whichever standard you choose, maintain the same scrolling increment throughout a window. The objective is to provide an increment that provides smooth but efficient scrolling. When a window cannot be scrolled any further in a particular direction, disable the scroll arrow corresponding to that direction.

Note The default system support for scroll bars does not disable the scroll arrow buttons when the region or area is no longer scrollable in this direction. However, it does provide support for you to disable the scroll arrow button under the appropriate conditions.

When scroll arrow buttons are pressed and held, they exhibit a special auto-repeat behavior. This action causes the window to continue scrolling in the associated direction as long as the pointer remains over the arrow button. If the pointer is moved off the arrow button while the user presses the mouse button, the auto-repeat behavior stops and does not continue unless the pointer is moved back over the arrow button (also when the pen tip is moved off the control).

Scroll Box

The scroll box, sometimes referred to as the elevator, thumb, or slider, moves along the scroll bar to indicate how far the visible portion is from the top (for vertical scroll bars) or from the left edge (for horizontal scroll bars). For example, if the current view is in the middle of a document, the scroll box in the vertical scroll bar is displayed in the middle of the scroll bar.

The size of the scroll box can vary to reflect the difference between what is visible in the window and the entire content of the file, as shown in Figure 6.8.

Note The proportional scroll box was not supported in earlier releases of Windows.

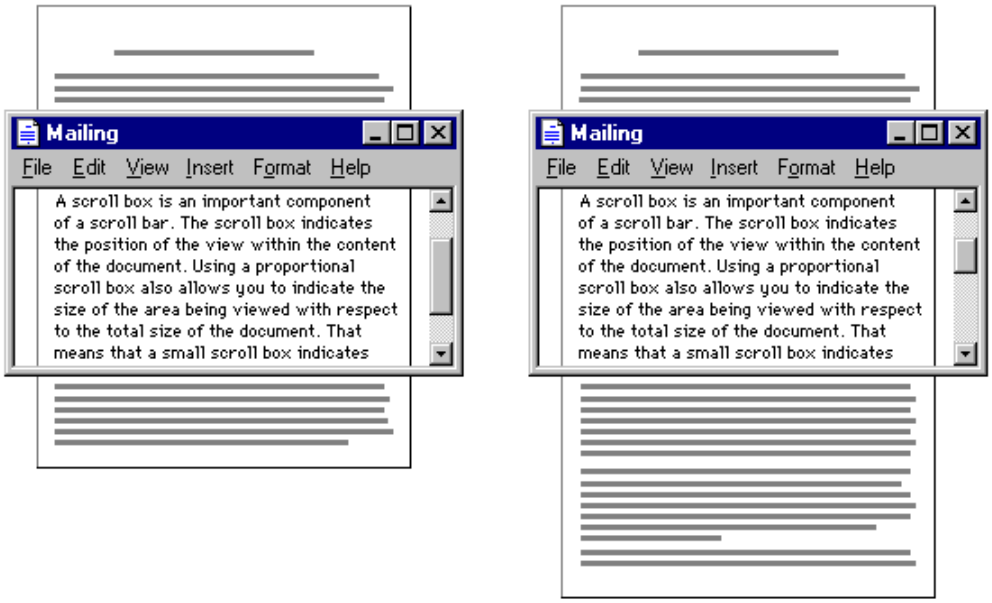


Figure 6.8 Proportional relationship between scroll box and content

For example, if the content of the entire document is visible in a window, the scroll box extends the entire length of the scroll bar, and the scroll arrows are disabled. Make the minimum size of the scroll box no smaller than the width of a window's sizing border.

The user can also scroll a window by dragging the scroll box. Update the view continuously as the user moves the scroll box. If you cannot support scrolling at a reasonable speed, you can scroll the information at the end of the drag operation as an alternative.

If the user starts dragging the scroll box and then moves outside of the scroll bar, the scroll box returns to its original position. The distance the user can move the pointer off the scroll bar before the scroll box snaps back to its original position is proportional to the width of the scroll bar. If dragging ends at this point, the scroll action is canceled—that is, no scrolling occurs. However, if the user moves the pointer back within the scroll-sensitive area, the scroll box returns to tracking the pointer movement. This behavior allows the user to scroll without having to remain within the scroll bar and to selectively cancel the initiation of a drag-scroll operation.

Dragging the scroll box to the end of the scroll bar implies scrolling to the end of that dimension; this does not always mean that the area cannot be scrolled further. If your application's document structure extends beyond the data itself, you can interpret dragging the scroll box to the end of its scroll bar as moving to the end of the data rather than the end of the structure. For example, the document of a typical spreadsheet exceeds the data in it — that is, the spreadsheet may have 65,000 rows, with data only in the first 50 rows. This means you can implement the scroll bar so that dragging the scroll box to the bottom of the vertical scroll bar scrolls to the last row containing data rather than the last row of the spreadsheet. The user can use the scroll arrow buttons to scroll further to the end of the structure. This situation also illustrates why disabling the scroll arrow buttons can provide important feedback so that the user can distinguish between scrolling to the end of data from scrolling to the end of the extent or structure. In the example of the spreadsheet, when the user drags the scroll box to the end of the scroll bar, the arrow would still be shown as enabled because the user can still scroll further, but it would be disabled when the user scrolls to the end of the spreadsheet.

Scroll Bar Shaft

The scroll bar shaft not only provides a visual context for the scroll box, it also serves as part of the scrolling interface. Clicking in the scroll bar shaft scrolls the view an equivalent size of the visible area in the direction of the click. For example, if the user clicks in the shaft below the scroll box in a vertical scroll bar, the view is scrolled a distance equivalent to the height of the view. Where possible, allow overlap from the previous view, as shown in Figure 6.9. For example, if the user clicks below the scroll box, the top line of the next screen becomes the line that was at the bottom of the previous screen. The same thing applies for clicking above the scroll box and horizontal scrolling. These conventions provide the user with a common reference point.

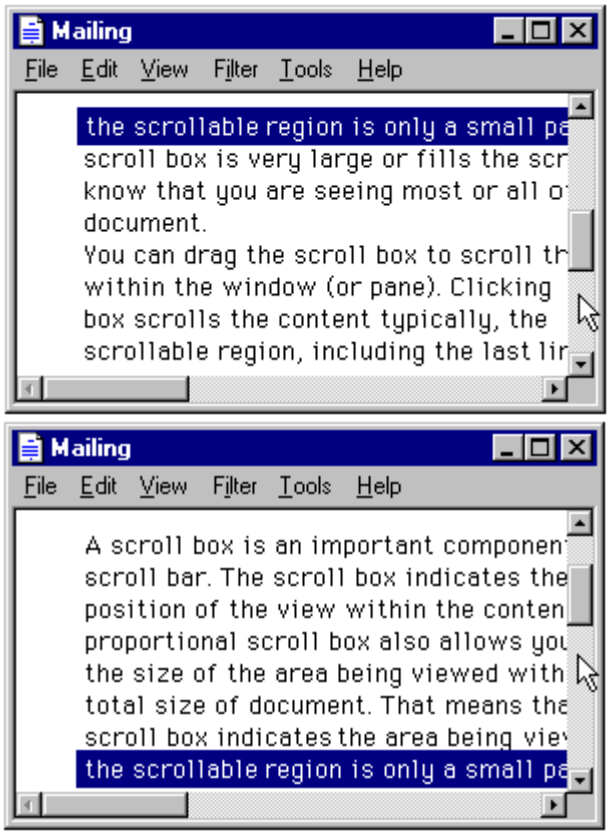


Figure 6.9 Scrolling with the scroll bar shaft by a screenful

Pressing and holding mouse button 1 with the pointer in the shaft auto-repeats the scrolling action. If the user moves the pointer outside the scroll-sensitive area while pressing the button, the scrolling action stops. The user can resume scrolling by moving the pointer back into the scroll bar area. (This behavior is similar to the effect of dragging the scroll box.)

Automatic Scrolling

The techniques previously summarized describe the explicit ways for scrolling. However, the user may also scroll as a secondary result of some situations. This type of scrolling is called *automatic scrolling*. The situations in which to support automatic scrolling are as follows:

- When the user begins or adjusts a selection and drags it past the edge of the scroll bar or window, scroll the area in the direction of the drag.
- When the user drags an object and approaches the edge of a scrollable area, scroll the area following the recommended auto-scroll conventions. Base the scrolling increment on the context of the destination and, if appropriate, on the size of the object being dragged.
- When the user enters text from the keyboard at the edge of a window or moves or copies an object into a location at the edge of a window, the view should scroll to allow the user to focus on the currently visible information. The amount to scroll depends on context. For example, for text typed in vertically, scroll a single line at a time. When scrolling horizontally, scroll in units greater than a single character to prevent continuous or uneven scrolling. Similarly, when the user transfers a graphic object near the edge of the view, base scrolling on the size of the object.
- If an operation results in a selection or moves the cursor, scroll the view to display the new selection. For example, for a Find command that selects a matching object, scroll the object into view because usually the user wants to focus on that location. In addition, other forms of navigation may cause scrolling. For example, completing an entry field in a form may result in navigating to the next field. In this case, if the field is not visible, the form can scroll to display it.

For more information about scrolling when the user drags objects, see Chapter 5, “General Interaction Techniques.”

Keyboard Scrolling

Use navigation keys to support scrolling with the keyboard. When the user presses a navigation key, the cursor moves to the appropriate location. For example, in addition to moving the cursor, pressing arrow keys at the edge of a scrollable area scrolls in the corresponding direction. Similarly, the PAGE UP and PAGE DOWN keys are comparable to clicking in the scroll bar shaft, but they also move the cursor.

Optionally, you can use the SCROLL LOCK key to facilitate keyboard scrolling. In this case, when the SCROLL LOCK key is toggled on and the user presses a navigation key, the view scrolls without affecting the cursor or selection.

Placing Adjacent Controls

It is sometimes convenient to locate controls or status bars adjacent to a scroll bar and position the end of the scroll bar to accommodate them. Although split box controls are an example of such controls, other types of controls also exist. Take care when placing adjacent elements; too many can make it difficult for users to scroll, particularly if you reduce the scroll bar too much. If you need a large number of controls, consider using a conventional toolbar instead.

For more information about toolbars, see Chapter 7, "Menus, Controls, and Toolbars."

Splitting Windows

A window can be split into two or more separate viewing areas, which are called *panes*. For example, a split window allows the user to examine two parts of a document at the same time. You can also use a split window to display different, yet simultaneous views of the same object (data), as shown in Figure 6.10.

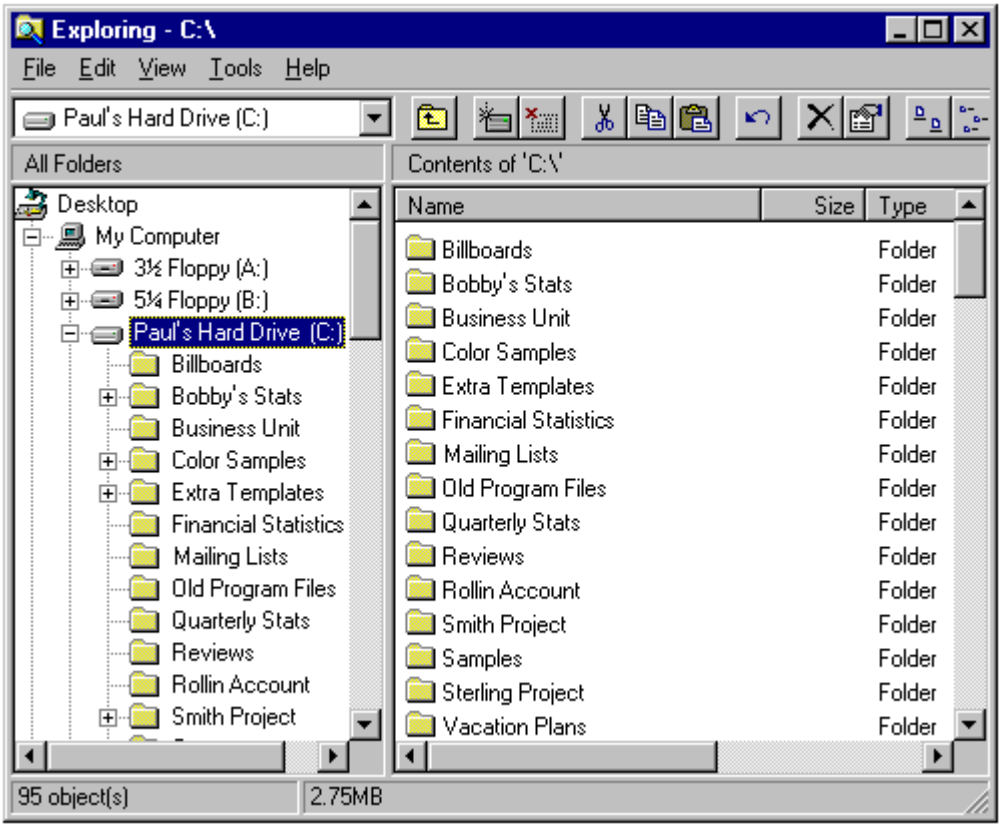


Figure 6.10 A split window

While you can use a split window panes to view the contents of multiple files or containers at the same time, displaying these in separate windows typically allows the user to better identify the files as individual elements. When you need to present views of multiple files as a single task, consider the window management techniques such as the Multiple Document Interface.

The panes that appear in a window can be implemented either as part of a window's basic design or as a user-configurable option. To support splitting a window that is not presplit by design, include a split box. A *split box* is a special control placed adjacent to the end of a scroll bar that splits or adjusts the split of a window. The size of the split box should be just large enough for the user to successfully target it with the pointer; the default size of a size handle, such as the window's sizing border, is a good guideline. Locate the split box at the top of the up arrow button of the vertical scroll bar or to the left of the left arrow button of a horizontal scroll bar as shown in Figure 6.11.

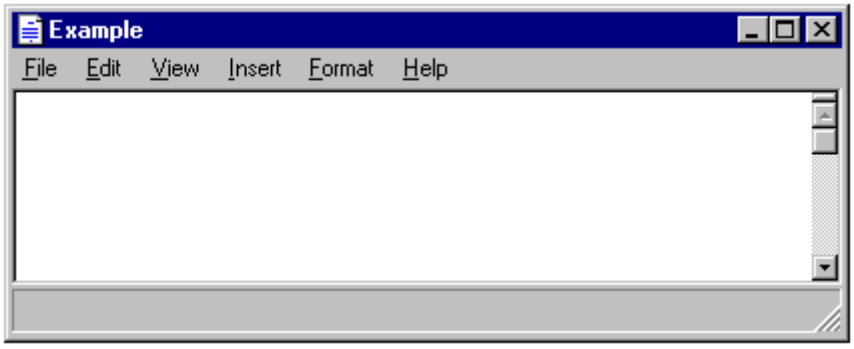


Figure 6.11 Split box location

The user splits a window by dragging the split box to the desired position. When the user positions the hot spot of the pointer over a split box, change the pointer's image to provide feedback and help the user target the split box. While the user drags the split box, move a representation of the split box and split bar with the pointer, as shown in Figure 6.12.

At the end of the drag, display a visual separator, called the *split bar*, that extends from one side of the window to the other, defining the edge between the resulting panes, as shown in Figure 6.12. Base the size for the split bar to be, at a minimum, the current setting for the size of window sizing borders. This allows you to appropriately adjust when a user adjusts size borders. If you display the split box after the split operation, place it adjacent to the split bar.

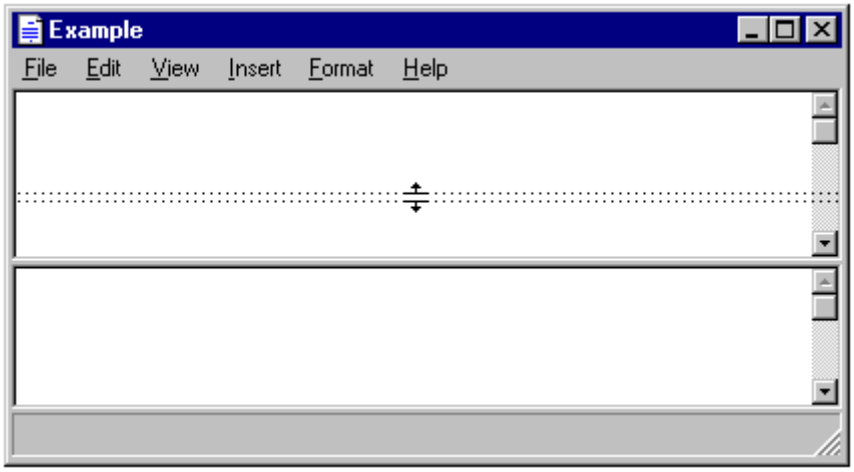


Figure 6.12 Moving the split bar

You can support dragging the split bar (or split box) to the end of the scroll bar to close the split. Optionally, you can also support double-clicking (or, for pens, double-tapping the screen) as a shortcut technique for splitting the window at some default location (for example, in the middle of the window or at the last split location) or for removing the split. This technique works best when the resulting window panes display peer views. It may not be appropriate when the design of the window requires that it always be displayed as split or for some types of specialized views.

To provide a keyboard interface for splitting the window, include a Split command for the window or view's menu. When the user chooses the Split command, split the window in the middle or in a context-defined location. Support arrow keys for moving the split box up or down; pressing the ENTER key sets the split at the current location. Pressing the ESC key cancels the split mode.

You can also use other commands to create a split window. For example, you can define specialized views that, when selected by the user, split a window to a fixed or variable set of panes. Similarly, you can enable the user to remove the split of a window by closing a view pane or by selecting another view command.

When the user splits a window, add scroll bars if the resulting panes require scrolling. In addition, you may need to scroll the information in panes so that the split bar does not obscure the content over which it appears. Scroll in the direction that is opposite of the split. Use a single scroll bar, at the appropriate side of the window, for a set of panes that scroll together. However, if the panes each require independent scrolling, a scroll bar should appear in each pane for that purpose. For example, the vertical scroll bars of a set of panes in a horizontally split window would typically be controlled separately.

When you use split window panes to provide separate views, independently maintain each pane's view properties, such as view type and selection state. Display only the selection in the active pane. However, if the selection state is shared across the panes, display a selection in all panes and support selection adjustment across panes.

When a window is closed, save the window's split state (that is, the number of splits, the place where they appear, the scrolled position in each split, and its selection state) as part of the view state information for that window so that it can be restored the next time the window is opened.

CHAPTER 7

Menus, Controls, and Toolbars

Microsoft Windows provides a number of interactive components that make it easier to carry out commands and specify values. These components also provide a consistent structure and set of interface conventions. This chapter describes the interactive elements of menus, controls, and toolbars, and how to use them.

Menus

Menus list the commands available to the user. By making commands visible, menus leverage user recognition rather than depending on user recollection of command names and syntax.

There are several types of menus, including drop-down menus, pop-up menus, and cascading menus. The following sections cover these menus in more detail.

The Menu Bar and Drop-down Menus

A *menu bar*, one of the most common forms of a menu, is a special area displayed across the top of a window directly below the title bar (as shown in Figure 7.1). A menu bar includes a set of entries called *menu titles*. Each menu title provides access to a *drop-down menu* composed of a collection of *menu items*, or choices.

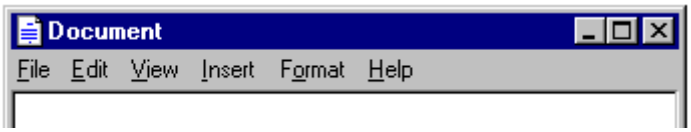


Figure 7.1 A menu bar

The content of the menu bar and its drop-down menus are determined by the functionality of your application and the context of a user's interaction. You can also optionally provide user configuration of the menu structure, including hiding the menu bar. If you provide this kind of option, supplement the interface with other components such as pop-up menus, handles, and toolbars, so that a user can access the functionality typically provided by the menu bar.

Drop-down Menu Interaction

When the user chooses a menu title, it displays its associated drop-down menu. To display a drop-down menu with the mouse, the user points to the menu title and presses or clicks mouse button 1. This action highlights the menu title and opens the menu. Tapping the menu title with a pen has the same effect as clicking the mouse.

If the user opens a menu by pressing the mouse button while the pointer is over the menu title, the user can drag the pointer over menu items in the drop-down menu. As the user drags, each menu item is highlighted, tracking the pointer as it moves through the menu. Releasing the mouse button with the pointer over a menu item chooses the command associated with that menu item and the system removes the drop-down menu. If the user moves the pointer off the menu and then releases the mouse button, the menu is "canceled" and the drop-down menu is removed. However, if the user moves the pointer back onto the menu before releasing the mouse button, the tracking resumes and the user can still select a menu item.

If the user opens a menu by clicking on the menu title, the menu title is highlighted and the drop-down menu remains displayed until the user clicks the mouse again. Clicking a menu item in the drop-down menu or dragging over and releasing the mouse button on a menu item chooses the command associated with the menu item and removes the drop-down menu.

The keyboard interface for drop-down menus uses the ALT key to activate the menu bar. When the user presses an alphanumeric key while holding the ALT key, or after the ALT key is released, the drop-down menu whose access key for the menu title matches the alphanumeric key (matching is not case sensitive) is displayed. Pressing a subsequent alphanumeric key chooses the menu item in the drop-down menu with the matching access character.

The user can also use arrow keys to access drop-down menus from the keyboard. When the user presses the ALT key, but has not yet selected a drop-down menu, LEFT ARROW and RIGHT ARROW keys highlight the previous or next menu title, respectively. At the end of the menu bar, pressing another arrow key in the corresponding direction wraps the highlight around to the other end of the menu bar. Pressing the ENTER key displays the drop-down menu associated with the selected menu title. If a drop-down menu is already displayed on that menu bar, then pressing LEFT ARROW or RIGHT ARROW navigates the highlight to the next drop-down menu in that direction, unless the drop-down menu has multiple columns, in which case the arrow keys move the highlight to the next column in that direction, and then to the next drop-down menu.

Pressing UP ARROW or DOWN ARROW in the menu bar also displays a drop-down menu if none is currently open. In an open drop-down menu, pressing these keys moves to the next menu item in that direction, wrapping the highlight around at the top or bottom. If the drop-down menu has multiple columns, then pressing the arrow keys first wraps the highlight around to the next column.

The user can cancel a drop-down menu by pressing the ALT key whenever the menu bar is active. This not only closes the drop-down menu, it also deactivates the menu bar. Pressing the ESC key also cancels a drop-down menu. However, the ESC key cancels only the current menu level. For example, if a drop-down menu is open, pressing ESC closes the drop-down menu, but leaves its menu title highlighted. Pressing ESC a second time unhighlights the menu title and deactivates the menu bar, returning input focus to the content information in the window.

You can assign shortcut keys to commands in drop-down menus. When the user presses a shortcut key associated with a command in the menu, the command is carried out immediately. Optionally, you can also highlight its menu title, but do not display the drop-down.

Common Drop-down Menus

This section describes the conventions for drop-down menus commonly used in applications. While these menus are not required for all applications, apply these guidelines when including these menus in your software's interface.

The File Menu

The File menu provides an interface for the primary operations that apply to a file. Your application should include commands such as Open, Save, Send To, and Print. These commands are often also included on the pop-up menu of the icon displayed in the title bar of the window.

For more information about the commands in the pop-up menu for a title bar icon, see the section, "Icon Pop-up Menus," later in this chapter.

If your application supports an Exit command, place this command at the bottom of the File menu preceded by a menu separator. When the user chooses the Exit command, close any open windows and files, and stop any further processing. If the object remains active even when its window is closed—for example, like a folder or printer—then include the Close command instead of Exit.

The Edit Menu

Include general purpose editing commands on the Edit menu. These commands include the Cut, Copy, and Paste transfer commands, OLE object commands, and the following commands (if they are supported).

For more information about menu commands for OLE objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Command	Function
Undo	Reverses last action.
Repeat	Repeats last action.
Find and Replace	Searches for and substitutes text.
Delete	Removes the current selection.
Duplicate	Creates a copy of the current selection.

Include these commands on this menu and on the pop-up menu of the selected object.

The View Menu

Commands on the View menu change the user's view of data in the window. Include commands on this menu that affect the view and not the data itself—for example, Zoom or Outline. Also include commands for controlling the display of particular interface elements in the view—for example, Show Ruler. These commands should be placed on the pop-up menu of the window or pane.

The Window Menu

Use the Window menu in multiple document interface-style (MDI) applications for managing the windows within an MDI workspace. Also include these commands on the pop-up menu of the parent MDI window.

For more information about the design of MDI software, see Chapter 9, "Window Management."

The Help Menu

The Help menu contains commands that provide access to Help information. Include a Help Topics command; this command provides access to the Help Topics browser, which displays topics included in your application's Help file. Alternatively, you can provide individual commands that access specific pages of the Help Topics browser such as Contents, Index, and Find Topic. You can also include other user assistance commands on this drop-down menu.

For more information about the Help Topics browser and support for user assistance, see Chapter 12, "User Assistance."

If you provide access to copyright and version information for your application, include an About *application name* command on this menu. When the user chooses this command, display a window containing the application's name, version number, copyright information, and any other informational properties related to the application. Display this information in a dialog box or alternatively as a copyright page of the property sheet of the application's main executable (.EXE) file. Do not use an ellipsis at the end of this command because the resulting window does not require the user to provide any further parameters.

Pop-up Menus

Even if you include a menu bar in your software's interface, you should also incorporate pop-up menus, as shown in Figure 7.2. Pop-up menus provide an efficient way for the user to access the operations of objects. Because pop-up menus are displayed at the pointer's current location, they eliminate the need for the user to move the pointer to the menu bar or a toolbar. In addition, because you populate pop-up menus with commands specific to the object or its immediate context, they reduce the number of commands the user must browse through. Pop-up menus also minimize screen clutter because they are displayed only upon demand and do not require dedicated screen space.

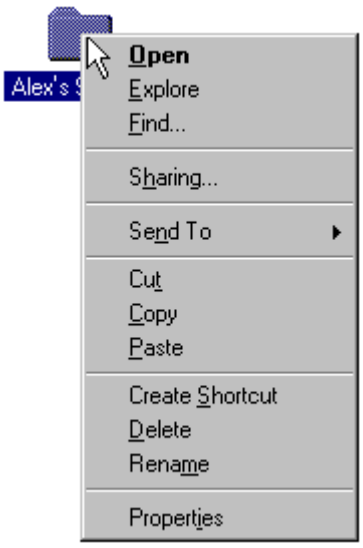


Figure 7.2 A pop-up menu

While a pop-up menu looks similar to a drop-down menu, a pop-up menu contains commands that apply to the selected object or objects and its context, rather than commands grouped by function. For example, a pop-up menu for a text selection can include the font properties of the text and the paragraph properties of which the selection is a part. However, keep the size of the pop-up menu as small as possible by limiting the items on the menu to common, frequent actions. It is better to include a single Properties command and allow the user to navigate among properties in the resulting property sheet than to list individual properties in the pop-up menu.

The container or the composition of which a selection is a part typically supplies the pop-up menu as a selection. Similarly, the commands included on a pop-up menu may not always be supplied by the object itself, but rather be a combination of those commands provided by the object and by its current container. For example, the pop-up menu for a file in a folder includes transfer commands. In this case, the folder (container) supplies the commands, not the files. Pop-up menus for OLE objects follow these same conventions.

For more information about the integration of commands for OLE objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Avoid using a pop-up menu as the exclusive means to a particular operation. At the same time, the items in a pop-up menu need not be limited only to commands that are provided in drop-down menus.

When ordering the commands in a pop-up menu, use the following guidelines:

- Place the object's primary commands first (for example, commands such as Open, Play, and Print), transfer commands, other commands supported by the object (whether provided by the object or by its context), and the What's This? command (when supported).
- Order the transfer commands as Cut, Copy, Paste, and other specialized Paste commands.
- Place the Properties command, when present, as the last command on the menu.

For more information about transfer commands and the Properties command, see Chapter 5, "General Interaction Techniques." For more information about the What's This? command, see Chapter 12, "User Assistance."

Pop-up Menu Interaction

With a mouse, the user displays a pop-up menu by clicking an object with button 2. The down transition of the mouse button selects the object. Upon the up transition, display the menu to the right and below the hot spot of the pointer; this is adjusted to avoid the menu being clipped by the edge of the screen.

If the pointer is over an existing selection when the user invokes a pop-up menu, display the menu that applies to that selection. If the menu is outside a selection but within the same selection scope, then establish a new selection (usually resetting the current selection in that scope) at the button down point and display the menu for the new selection. If the user clicks the button a second time within the same selection, remove the menu. Also, dismiss the pop-up menu when the user clicks outside the menu with button 1 or if the user presses the ESC key.

You can support pop-up menus for objects that are implicitly selected or cannot be directly selected, such as scroll bars or items in a status bar. When providing pop-up menus for objects such as controls, include commands for the object that the control represents, rather than for the control itself. For example, a scroll bar represents a navigational view of a document, so commands might include Beginning of Document, End of Document, Next Page, and Previous Page. But when a control represents itself as an object, as in a forms layout or window design environment, you can include commands that apply to the control—for example, commands to move or copy the control.

The pen interface uses an action handle in pen-enabled controls to access the pop-up menu for the selection. Tapping the action handle displays the pop-up menu, as shown in Figure 7.3.

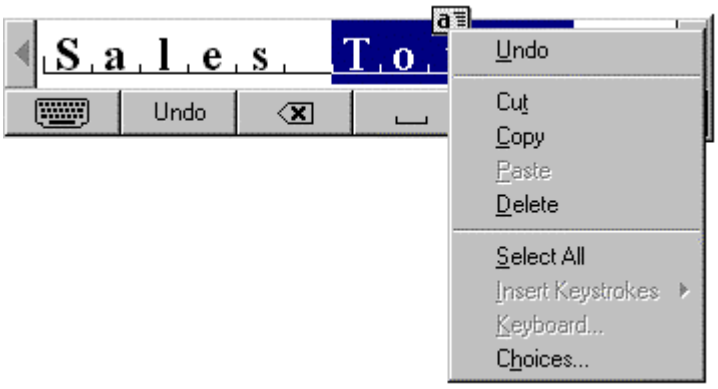


Figure 7.3 Pop-up menus with the pen interface

In addition, you can use techniques like barrel-tapping or the pop-up menu gesture to display a pop-up menu. This interaction is equivalent to a mouse button 2 click.

For more information about pen interaction techniques, see Chapter 5, "General Interaction Techniques."

As for the keyboard interface for displaying a pop-up menu for a selection, support SHIFT+F10 and the "application key" for keyboards that support the Microsoft Logo keys specification. In addition, menu access keys, arrow keys, ENTER, and ESC keys all operate in the same fashion in the menu as they do in drop-down menus. To enhance space and visual efficiency, avoid including shortcut keys in pop-up menus.

Common Pop-up Menus

The pop-up menus included in any application depend on the objects and context supplied by that application. The following sections describe common pop-up menus for all Windows-based applications.

The Window Pop-up Menu

The window pop-up menu is the pop-up menu associated with the window—do not confuse it with the Window drop-down menu found in MDI applications. The window pop-up menu replaces the Windows 3.1 Control menu, also referred to as the System menu. For example, a typical primary window includes Close, Restore, Move, Size, Minimize, and Maximize.

You can also include other commands on the window's menu that apply to the window or the view within the window. For example, an application can append a Split command to the menu to facilitate splitting the window into panes. Similarly, you can add commands that affect the view, such as Outline, commands that add, remove, or filter elements from the view, such as Show Ruler, or commands that open certain subordinate or special views in secondary windows, such as Show Color Palette.

A secondary window also includes a pop-up menu. Usually, because the range of operations are more limited than in a primary window, a secondary window's pop-up menu includes only Move and Close commands, or just Move. Palette windows can also include an Always on Top command that sets the window to always be on top of its parent window and secondary windows of its parent window.

The user displays a window's pop-up menu by clicking mouse button 2 anywhere in the title bar area, excluding the title bar icon. Clicking on the title bar icon displays the pop-up menu for the icon. For the pen, performing barrel-tapping or the equivalent pop-up menu gesture on these areas displays the menu. Pressing ALT+SPACEBAR also displays the menu.

Note To support compatibility with previous versions of Windows, the system also supports clicking button 1 on the icon in the title bar to access the pop-up menu of a window.

Icon Pop-up Menus

Pop-up menus displayed for icons include operations of the objects represented by those icons. Accessing the pop-up menu of an application or document icon follows the standard conventions for pop-up menus, such as displaying the menus with a mouse button 2 click.

An icon's container application supplies the pop-up menu for the icon. For example, pop-up menus for icons placed in standard folders or on the desktop are automatically provided by the system. However, your application supplies the pop-up menus for OLE embedded or linked objects placed in it—that is, placed in the document or data files your application supports.

For more information about supporting pop-up menus for OLE objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

The container populates the pop-up menu for an icon with commands the container supplies for its content, such as transfer commands and those registered by the object's type. For example, an application can register a New command that automatically generates a new data file of the type supported by the application.

For more information about registering commands, see Chapter 10, "Integrating with the System."

The pop-up menu of an application's icon, for example, the Microsoft WordPad executable file, typically includes the following commands.

Table 7.1 Application File Icon Pop-up Menu Commands

Command	Meaning
Open	Opens the application file.
Send To	Displays a submenu of destinations to which the file can be transferred. The content of the submenu is based on the content of the system's Send To folder.
Cut	Marks the file for moving. (Registers the file on the Clipboard.)
Copy	Marks the file for duplication. (Registers the file on the Clipboard.)
Paste	Attempts to open the file registered on the Clipboard with the application.
Create Shortcut	Creates a shortcut icon of the file.
Delete	Deletes the file.
Rename	Allows the user to edit the filename.
Properties	Displays the properties for the file.

An icon representing a document or data file typically includes the following common menu items for the pop-up menu for its icon.

Table 7.2 Document or Data File Icon Pop-up Menu Commands

Command	Meaning
Open	Opens the file's primary window.
Print	Prints the file on the current default printer.
Send To	Displays a submenu of destinations to which the file can be transferred. The content of the submenu is based on the content of the system's Send To folder.
Cut	Marks the file for moving. (Registers the file on the Clipboard.)
Copy	Marks the file for duplication. (Registers the file on the Clipboard.)
Delete	Deletes the file.
Rename	Allows the user to edit the filename.
Properties	Displays the properties for the file.

For the Open and Print commands to appear on the menu, your application must register these commands in the system registry. You can also register additional or replacement commands. For example, you can optionally register a Quick View command that displays the content of the file without running the application and a What's This? command that displays descriptive information for your data file types.

For more information about registering commands and the Quick View command, see Chapter 10, "Integrating with the System." For more information about the What's This? command, see Chapter 12, "User Assistance."

The icon in the title bar of a window represents the same object as the icon the user opens. As a result, the application associated with the icon also includes a pop-up menu with appropriate commands for the title bar's icon. When the icon of an application appears in the title bar, include the same commands on its pop-up menu as are included for the icon that the user opens, unless a particular command cannot be applied when the application's window is open. In addition, replace the Open command with Close.

Similarly, when the icon of the data or document file appears in the title bar, you also use the same commands as found on its file icon, with the following exceptions: replace the Open command with a Close command and add Save if the edits in the document require explicit saving to file.

For an MDI application, supply a pop-up menu for the application icon in the parent window, following the conventions for application title bar icons. Include the following commands where they apply.

Table 7.3 Optional MDI Parent Window Title Bar Icon Pop-up Menu Commands

Command	Meaning
New	Creates a new data file or displays a list of data file types supported by the application from which the user can choose.
Save All	Saves all data files open in the MDI workspace, and the state of the MDI window.
Find	Displays a window that allows the user to specify criteria to locate a data file.

For more information about the design of MDI-style applications, see Chapter 9, "Window Management."

In addition, supply an appropriate pop-up menu for the title bar icon that appears in the child window's title bar. You can follow the same conventions for non-MDI data files.

Cascading Menus

A *cascading menu* (also referred to as a *hierarchical menu* or child menu) is a submenu of a menu item. The visual cue for a cascading menu is the inclusion of a triangular arrow display adjacent to the label of its parent menu item.

You can use cascading menus to provide user access to additional choices rather than taking up additional space in the parent menu. They may also be useful for displaying hierarchically related objects.

Be aware that cascading menus can add complexity to the menu interface by requiring the user to navigate further through the menu structure to get to a particular choice. Cascading menus also require more coordination to handle the changes in direction necessary to navigate through them.

In light of these design tradeoffs, use cascading menus sparingly. Minimize the number of levels for any given menu item, ideally limiting your design to a single submenu. Avoid using cascading menus for frequent, repetitive commands.

As an alternative, make choices available in a secondary window, particularly when the choices are independent settings; this allows the user to set multiple options in one invocation of a command. You can also support many common options as entries on a toolbar.

The user interaction for a cascading menu is similar to that of a drop-down menu from the menu bar, except a cascading menu displays after a short time-out. This avoids the unnecessary display of the menu if the user is browsing or navigating to another item in the parent menu. Once displayed, if the user moves the pointer to another menu item, the cascading menu is removed after a short time-out. This time-out enables the user to directly drag from the parent menu into an entry in its cascading menu.

Menu Titles

All drop-down and cascading menus have a menu title. For drop-down menus, the menu title is the entry that appears in the menu bar. For cascading menus, the menu title is the name of the parent menu item. Menu titles represent the entire menu and should communicate as clearly as possible the purpose of all items on the menu.

Use single words for menu bar menu titles. Multiple word titles or titles with spaces may be indistinguishable from two one-word titles. In addition, avoid uncommon compound words, such as Fontsize.

Define one character of each menu title as its access key. This character provides keyboard access to the menu. Windows displays the access key for a menu title as an underlined character, as shown in Figure 7.4.

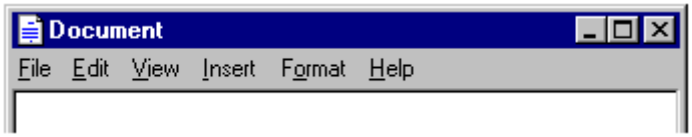


Figure 7.4 Access keys in a menu bar

Define unique access keys for each menu title. Using the same access key for more than one menu title may eliminate direct access to a menu.

For more information about keyboard input and defining access keys, see Chapter 4, "Input Basics."

Menu Items

Menu items are the individual choices that appear in a menu. Menu items can be words, graphics—such as icons—or graphics and word combinations that represent the actions presented in the menu, as shown in Figure 7.5. The format for a menu item provides the user with visual cues about the nature of the effect it represents.

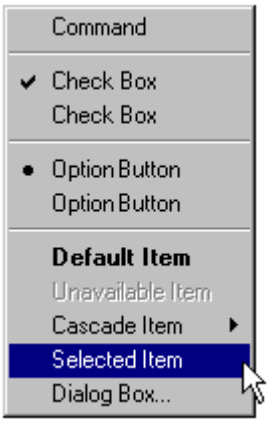


Figure 7.5 Different menu items

Whenever a menu contains a set of related menu items, you can separate those sets with a grouping line known as a *separator*. The standard separator is a single line that spans the width of the menu. Avoid using menu items themselves as group separators, as shown in Figure 7.6.

Figure 7.6 Inappropriate use of a separator

Always provide the user with a visual indication about which menu items can be applied. If a menu item is not appropriate or applicable in a particular context, then disable or remove it. Leaving the menu item enabled and presenting a message box when the user selects the menu item is an ineffective method for providing feedback.

It is better to disable a menu item rather than remove it because this provides more stability in the interface. However, if the context is such that the menu item is no longer or never relevant, remove it. For example, if a menu displays a set of open files and one of those files is closed or deleted, it is appropriate to remove the corresponding menu item.

If all items in a menu are disabled, disable its menu title. If you disable a menu item or its title, it does not prevent the user from browsing or choosing it. If you provide status bar messages, display a message indicating that the command is unavailable and why.

For more information about status bar messages, see Chapter 12, "User Assistance."

The system provides a standard appearance for displaying disabled menu items. If you are supplying your own visuals for a disabled menu item, follow the visual design guidelines for how to display it with unavailable appearance.

For more information about displaying commands with an unavailable appearance, see Chapter 13, "Visual Design."

Types of Menu Items

Many menu items take effect as soon as they are chosen. If the menu item is a command that requires additional information prior to execution, follow the command with an *ellipsis* (...). The ellipsis informs the user that information is incomplete. When it is used with a command, it indicates that the user needs to provide more information to complete that command. Such commands usually result in the display of a dialog box. For example, the Save As command includes an ellipsis because the command is not complete until the user supplies or confirms a filename.

Not every command that produces a dialog box or other secondary window needs to be listed with an ellipsis. For example, the Properties command does not have an ellipsis after it because executing the Properties command displays a properties window. After completing the command, no further parameters or actions are required to fulfill the intent of the command. Similarly, do not include an ellipsis for a command that can result in the display of a message box.

While you can use menu items to carry out commands, you can also use menu items to switch a mode or set a state or property, rather than initiating a process. For example, choosing an item from a menu that contains a list of tools or views implies changing to that state. If the menu item represents a property value, when the user chooses the menu item, the property setting changes.

Menu items for state settings can be independent or interdependent:

- Independent settings are the menu equivalent of check boxes. For example, if a menu contains text properties, such as Bold and Italic, they form a group of independent settings. The user can change each setting without affecting the others, even though they both apply to a single text selection. Include a check mark to the left of an independent setting when that state applies.
- Interdependent settings are the menu equivalent of option buttons. For example, if a menu contains alignment properties such as Left, Center, and Right, they form a group of interdependent settings. Because a particular paragraph can have only one type of alignment, choosing one resets the property to be the chosen menu item setting. When the user chooses an interdependent setting, place an option button mark to the left of that menu item.

In some cases, it is appropriate to reflect the change of a setting by changing the menu item, rather than adding a graphic. If the two states of a setting are obvious opposites, such as the presence or absence of a property value, use a check mark. For example, when reflecting the state of a text selection with a menu item labeled Bold, show a check mark next to the menu item when the text selection is bold and no check mark when it is not. If a selection contains mixed values for the same state reflected in the menu, you also display the menu without the check mark.

If the two states of the setting are not obvious opposites, use a pair of alternating menu item names to indicate the two states. For example, a naive user might guess that the opposite of a menu item called Full Duplex is Empty Duplex. Because of this ambiguity, pair the command with the alternative name Half Duplex, rather using a graphic to indicate the alternative states, and consider the following guidelines for how to display those alternatives:

- If there is room in a menu, include both alternatives as individual menu items and interdependent choices. This avoids confusion because the user can view both options simultaneously. You can also use menu separators to group the choices.
- If there is not sufficient room in the menu for the alternative choices, you can use a single menu item and change its name to the alternative action when selected. In this case, the menu item's name does not reflect the current state; it indicates the state after choosing the item. Where possible, define names that use the same access key. For example, the letter D could be used for a menu item that toggles between Full Duplex and Half Duplex.

Note Avoid defining menu items that change depending on the state of a modifier key. Such techniques hide functionality from a majority of users.

A menu can also have a default item. A default menu item reflects a choice that is also supported through a shortcut technique, such as double-clicking or drag and drop. For example, if the default command for an icon is Open, display this as the default menu item. Similarly, if the default command for a drag and drop operation is Copy,

display this command as the default menu item in the pop-up menu that results from a nondefault drag and drop operation (button 2). The system default appearance for designating a default menu item is to display the command as bold text.

For more information about default operations, see Chapter 5, "General Interaction Techniques."

Menu Item Labels

Include a descriptive text or graphic label for each menu item. Even if you provide a graphic for the label, consider including text as well. The text allows you to provide more direct keyboard access to the user.

Use the following guidelines for writing menu items:

- Define unique item names within a menu. However, item names can be repeated in different menus to represent similar or different actions.
- Use a single word or multiple words, but keep the wording brief and succinct. Verbose menu item names can make it harder for the user to scan the menu.
- Define unique access keys for each menu item within a menu. This provides the user direct keyboard access to the menu item. The guidelines for selecting an access key for menu items are the same as for menu titles, except that the access key for a menu item can also be a number included at the beginning of the menu item name. This is useful for menu items that vary, such as filenames. Where possible, also define consistent access keys for common commands.

For more information about defining access keys, see Chapter 4, "Input Basics." For more information about common access key assignments, see Appendix B, "Keyboard Interface Summary."

- Follow book title capitalization rules for menu item names. Capitalize the first letter of every word, except for articles, conjunctions, and prepositions that occur other than at the beginning or end of a multiple-word name. For example, the following menu names are correct: New Folder, Go To, Select All, and Table of Contents.
- Avoid formatting individual menu item names with different text properties. Even though these properties illustrate a particular text style, they also may make the menu cluttered, illegible, or confusing. For example, it may be difficult to indicate an access key if an entire menu entry is underlined.

Shortcut Keys in Menu Items

A drop-down menu item can also display a keyboard shortcut associated with the command. Display the shortcut key on the menu next to the item and align shortcuts with other shortcuts in the menu. Typically, they are left aligned at the first tab position after the longest item in the menu that has a shortcut. Do not use spaces for alignment because they may not display properly in the proportional font used by the system to display menu text or when the font setting menu text changes.

You can match key names with those commonly inscribed on the keycap. Display CTRL and SHIFT key combinations as Ctrl+*key* (rather than Control+*key* or CONTROL+*key* or ^+*key*) and Shift+*key*. When using function keys for menu item shortcuts, display the name of the key as F*n*, where *n* is the function key number.

For more information about the selection of shortcut keys, see Chapter 4, "Input Basics."

Avoid including shortcut keys in pop-up menus. Pop-up menus are already a shortcut form of interaction and are typically accessed with the mouse. In addition, excluding shortcut keys makes pop-up menus easier for users to scan.

Controls

Controls are graphic objects that represent the properties or operations of other objects. Some controls display and allow editing of particular values. Other controls start an associated command.

Each control has a unique appearance and operation designed for a specific form of interaction. The system also provides support for designing your own controls. When defining your own controls, follow the conventions consistent with those provided by the system-supplied controls.

For more information about using standard controls and designing your own controls, see Chapter 13, "Visual Design."

Like most elements of the interface, controls provide feedback indicating when they have the input focus and when they are activated. For example, when the user interacts with controls using a mouse, each control indicates its selection upon the down transition of the mouse button, but does not activate until the user releases the button, unless the control supports auto-repeat.

Controls are generally interactive only when the pointer, also referred to as the hot spot, is over the control. If the user moves the pointer off the control, the control no longer responds to the input device. For some controls, such as scroll bars, the pointer is inactive when it is outside a defined hot zone of the control. If the user moves the pointer back onto the control, it once again responds to the input device.

Many controls provide labels. Because labels help identify the purpose of a control, it is best to label a control. If a control does not have a label, you can provide a label using a static text field or a tooltip control. Define an access

key for text labels to provide the user direct keyboard access to a control. Where possible, define consistent access keys for common commands.

For more information about access key assignments for common commands, see Appendix B, "Keyboard Interface Summary."

While controls provide specific interfaces for user interaction, you can also include pop-up menus for controls. This can provide an effective way to transfer the value the control represents or to provide access to context-sensitive Help information. The interface to pop-up menus for controls follows the standard conventions for pop-up menus, except that it does not affect the state of the control; that is, clicking the control with button 2 does not start the control. The only action is the display of the pop-up menu.

A pop-up menu for a control is contextual to what the control represents, rather than the control itself. Therefore, do not include commands such as Set, Unset, Check, or Uncheck. The exception is in a forms design or window layout context, where the commands on the pop-up menu can apply to the control itself.

Buttons

Buttons are controls that start actions or change properties. There are three basic types of buttons: command buttons, option buttons, and check boxes.

Command Buttons

A *command button*, also referred to as a push button, is a control, commonly rectangular in shape, that includes a label (text, graphic, or sometimes both), as shown in Figure 7.7.

Figure 7.7 Command buttons

When the user chooses a command button with mouse button 1 (for pens, tapping), the command associated with the button is carried out. When the user presses the mouse button, the input focus moves to the button, and the button state changes to its pressed appearance. If the user moves the pointer off the command button while the mouse button remains pressed, the button returns to its original state. Moving the pointer back over the button while pressing the mouse button returns the button to its pressed state.

When the user releases the mouse button with the pointer on the command button, the command associated with the control starts. If the pointer is not on the control when the user releases the mouse button, no action occurs.

You can define access keys and shortcut keys for command buttons. In addition, you can use the TAB key and arrow keys to support user navigation to or between command buttons. The SPACEBAR activates a command button if the user moves the input focus to the button.

For more information about navigation and activation of controls, see Chapter 8, "Secondary Windows."

The effect of choosing a button is immediate with respect to its context. For example, in toolbars, clicking a button carries out the associated action. In a secondary window, such as a dialog box, activating a button may initiate a transaction within the window, or apply a transaction and close the window.

The command button's label represents the action the button starts. When using a text label, the text should follow the same capitalization conventions defined for menus. If the control is unavailable, the label of the button is displayed as unavailable.

Include an ellipsis (...) as a visual cue for buttons associated with commands that require additional information. Like menu items, the use of an ellipsis indicates that further information is needed, not simply that a window will appear. Some buttons, when clicked, can display a message box, but this does not imply that the command button's label should include an ellipsis.

You can use command buttons to enlarge a secondary window and display additional options, also known as an *unfold button*. An unfold button is not really a different type of control, but the use of a command button for this specific function. When using a command button for this purpose, include a pair of "greater than" (>>) characters as part of the button's label.

For more information about the use of an ellipsis in a button label and unfold buttons in secondary windows, see Chapter 8, "Secondary Windows."

In some cases, a command button can represent an object and its default action. For example, the taskbar buttons represent an object's primary window and the Restore command. When the user clicks on the button with mouse button 1, the default command of the object is carried out. Clicking on the button with mouse button 2 displays a pop-up menu associated with the object.

You can also use command buttons to reflect a mode or property value similar to the use of option buttons or check boxes. While the typical interaction for a command button is to return to its normal "up" state, if you use it to represent a state, display the button in the option-set appearance, which is a checkerboard pattern in the button's highlight color on the background of the button, as shown in Table 7.4.

Table 7.4 Command Button Appearance

Visual	Representation
	Normal appearance
	Pressed appearance
	Option-set appearance
	Unavailable appearance
	Option-set, unavailable appearance
	Mixed-value appearance
	Input focus appearance

For more information about the appearance of different states of buttons, see Chapter 13, "Visual Design."

You can also use command buttons to set tool modes—for example, in drawing or forms design programs for drawing out specific shapes or controls. In this case, design the button labels to reflect the tool's use. When the user chooses the tool (that is, clicks the button), the button is displayed using the option-set appearance and the pointer is changed to indicate the change of the mode of interaction.

You can also use a command button to display a pop-up menu. This convention is known as a *menu button*. While this is not a specific control provided by the system, you can create this interface using the standard components.

A menu button looks just like a standard command button, except that, as a part of its label, it includes a triangular arrow similar to the one found in cascading menu titles, as shown in Figure 7.8.

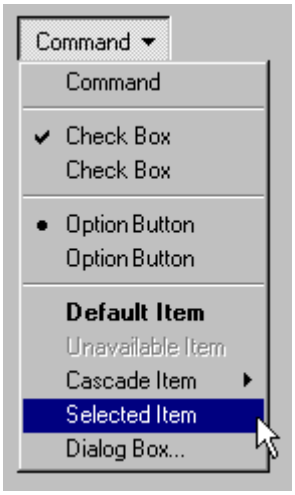


Figure 7.8 A menu button

A menu button supports the same type of interaction as a drop-down menu; the menu is displayed when the button is pressed and allows the user to drag into the menu from the button and make menu selections. Like any other menu, use highlighting to track the movement of the pointer.

Similarly, when the user clicks a menu button, the menu is displayed. At this point, interaction with the menu is the same as with any menu. For example, clicking a menu item carries out the associated command. Clicking outside the menu or on the menu button removes the menu.

When pressed, display the menu button with the pressed appearance. When the user releases the mouse button and the menu is displayed, the command button uses the option-set appearance—a checkerboard appearance using the button's highlight color on the background of the button. Otherwise, the menu button's appearance is the same as a typical command button. For example, if the button is unavailable, the button displays the unavailable appearance.

For more information about the appearance of button states, see Chapter 13, "Visual Design."

Option Buttons

An *option button*, also referred to as a radio button, represents a single choice within a limited set of mutually exclusive choices—that is, in any group of option buttons, only one option in the group can be set. Accordingly, always group option buttons in sets of two or more, as shown in Figure 7.9.

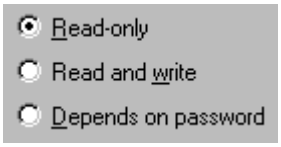


Figure 7.9 A set of option buttons

Option buttons appear as a set of small circles. When an option button choice is set, a dot appears in the middle of the circle. When the choice is not the current setting, the circle is empty. Avoid using option buttons to start an action other than the setting of a particular option or value represented by the option button. The only exception is that you can support double-clicking the option button as a shortcut for setting the value and carrying out the default command of the window in which the option buttons appear, if choosing an option button is the primary user action for the window.

You can use option buttons to represent a set of choices for a particular property. When the option buttons reflect a selection with mixed values for that property, display all the buttons in the group using the mixed-value appearance to indicate that multiple values exist for that property. The mixed-value appearance for a group of option buttons displays all buttons without a setting dot, as shown in Figure 7.10.

Figure 7.10 Option buttons with mixed-value appearance

If the user chooses any option button in a group with mixed-value appearance, that value becomes the setting for the group; the dot appears in that button and all the other buttons in the group remain empty.

Limit the use of option buttons to small sets of options, typically seven or less, but always at least two. If you need more choices, consider using another control, such as a single selection list box or drop-down list box.

Option buttons include a text label. (If you need graphic labels for a group of exclusive choices, consider using command buttons instead.) Define the label to best represent the value or effect for that choice. Also use the label to indicate when the choice is unavailable. Use sentence capitalization for an option button's label; only capitalize the first letter of the first word, unless it is a word in the label normally capitalized.

For more information about labeling or appearance states, see Chapter 13, "Visual Design."

Because option buttons appear as a group, you can use a group box control to visually define the group. You can label the option buttons to be relative to a group box's label. For example, for a group box labeled Alignment, you can label the option buttons as Left, Right, and Center.

As with command buttons, the mouse interface for choosing an option button uses a click with mouse button 1 (for pens, tapping) either on the button's circle or on the button's label. The input focus is moved to the option button when the user presses the mouse button, and the option button displays its pressed appearance. If the user moves the pointer off the option button before releasing the mouse button, the option button is returned to its original state. The option is not set until the user releases the mouse button while the pointer is over the control. Releasing the mouse button outside of the option button or its label has no effect on the current setting of the option button. In addition, successive mouse clicks on the same option button do not toggle the button's state; the user needs to explicitly select an alternative choice in the group to change or restore a former choice.

Assign access keys to option button labels to provide a keyboard interface to the buttons. You can also define the TAB or arrow keys to allow the user to navigate and choose a button.

For more information about the guidelines for defining access keys, see Chapter 4, "Input Basics." For more information about navigation and interaction with option buttons, see Chapter 8, "Secondary Windows."

Check Boxes

Like option buttons, check boxes support options that are either on or off; check boxes differ from option buttons in that you typically use check boxes for independent or nonexclusive choices. A check box appears as a square box with an accompanying label. When the choice is set, a check mark appears in the box. When the choice is not set, the check box is empty, as shown in Figure 7.11.

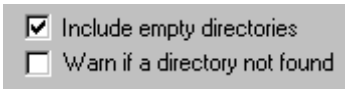


Figure 7.11 A set of check boxes

As in the case of independent settings in menus, use check boxes only when both states of the choice are clearly opposite and unambiguous. If this is not the case, then use option buttons or some other form of single selection choice control instead.

A check box's label is typically in text and the standard control includes a label. (Use a command button instead of a check box when you need a nonexclusive choice with a graphic label.) Define the label to appropriately express the value or effect of the choice. Use sentence capitalization for multiple word labels. The label also serves as an indication of when the control is unavailable.

Group related check box choices. If you group check boxes, it does not prevent the user from setting the check boxes on or off in any combination. While each check box's setting is typically independent of the others, you can use a check box's setting to affect other controls. For example, you can use the state of a check box to filter the content of a list. If you have a large number of choices or if the number of choices varies, use a multiple selection list box instead of check boxes.

When the user clicks a check box with mouse button 1 (for pens, tapping) either on the check box square or on the check box's label, that button is chosen and its state is toggled. When the user presses the mouse button, the input focus moves to the control and the check box assumes its pressed appearance. Like option buttons and other controls, if the user moves the pointer off the control while holding down the mouse button, the control's appearance returns to its original state. The setting state of the check box does not change until the mouse button is released. To change the control's setting, the pointer must be over the check box or its label when the user releases the mouse button.

Define access keys for check box labels to provide a keyboard interface for navigating to and choosing a check box. In addition, the TAB key and arrow keys can also be supported to provide user navigation to or between check boxes. In a dialog box, for example, the SPACEBAR toggles a check box when the input focus is on the check box.

For more information about the guidelines for defining access keys, see Chapter 4, "Input Basics." For more information about navigation and choosing controls with the keyboard, see Chapter 8, "Secondary Windows."

If you use a check box to display the value for the property of a multiple selection whose values for that property differ (for example, for a text selection that is partly bold), display the check box in its mixed-value appearance, a checkerboard pattern inside the box, as shown in Figure 7.12.

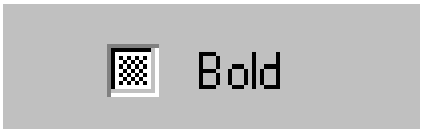


Figure 7.12 A mixed-value check box (magnified)

If the user chooses a check box in the mixed-value state, the associated value is set and a check mark is placed in it. This implies that the property of all elements in the multiple selection will be set to this value when it is applied. If the user chooses the check box again, the setting to be unchecked is toggled. If applied to the selection, the value will not be set. If the user chooses the check box a third time, the value is toggled back to the mixed-value state. When the user applies the value, all elements in the selection retain their original value. This three-state toggling occurs only when the control represents a mixed set of values.

List Boxes

A *list box* is a convenient, preconstructed control for displaying a list of choices for the user. The choices can be text, color, icons, or other graphics. The purpose of a list box is to display a collection of items and, in most cases, support selection of a choice of an item or items in the list.

List boxes are best for displaying large numbers of choices that vary in number or content. If a particular choice is not available, omit the choice from the list. For example, if a point size is not available for the currently selected font, do not display that size in the list.

Order entries in a list using the most appropriate choice to represent the content in the list and to facilitate easy user browsing. For example, alphabetize a list of filenames, but put a list of dates in chronological order. If there is no natural or logical ordering for the content, use ascending or alphabetical ordering—for example, 0–9 or A–Z.

List box controls do not include their own labels. However, you can include a label using a static text field; the label enables you to provide a descriptive reference for the control and keyboard access to the control. Use sentence capitalization for multiple word labels and make certain that your support for keyboard access moves the input focus to the list box and not the static text field label.

For more information about navigation to controls in a secondary window, see Chapter 8, "Secondary Windows." For more information about defining access keys for control labels, see Chapter 4, "Input Basics." For more information about static text fields, see the section, "Static Text Fields," later in this chapter.

When a list box is disabled, display its label using an unavailable appearance. If possible, display all of the entries in the list as unavailable to avoid confusing the user as to whether the control is enabled or not.

The width of the list box should be sufficient to display the average width of an entry in the list. If that is not practical because of space or the variability of what the list might include, consider one or more of the following options:

- Make the list box wide enough to allow the entries in the list to be sufficiently distinguished.
- Use an ellipsis (...) in the middle or at the end of long text entries to shorten them, while preserving the important characteristics needed to distinguish them. For example, for long paths, usually the beginning and end of the path are the most critical; you can use an ellipsis to shorten the entire name: \Sample\...\Example.

- Include a horizontal scroll bar. This option reduces some usability, because adding the scroll bar reduces the number of entries the user can view at one time. In addition, if most entries in the list box do not need to be horizontally scrolled, including a horizontal scroll bar accommodates the infrequent case.

When the user clicks an item in a list box, it becomes selected. Support for multiple selection depends on the type of list box you use. List boxes also include scroll bars when the number of items in the list exceeds the visible area of the control.

Arrow keys also provide support for selection and scrolling a list box. In addition, list boxes include support for keyboard selection using text keys. When the user presses a text key, the list navigates and selects the matching item in the list, scrolling the list if necessary to keep the user's selection visible. Subsequent key presses continue the matching process. Some list boxes support sequential matches based on timing; each time the user presses a key, the control matches the next character in a word if the user presses the key within the system's time-out setting. If the time-out elapses, the control is reset to matching upon the first character. Other list box controls, such as combo boxes and drop-down combo boxes, do sequential character matching based on the characters typed into the text box component of the control. These controls may be preferable because they do not require the user to master the timing sequence. However, they do take up more space and potentially allow the user to type in entries that do not exist in the list box.

When the list is scrolled to the beginning or end of data, disable the corresponding scroll bar arrow button. If all items in the list are visible, disable both scroll arrows. If the list box never includes more items that can be shown in the list box, so that the user will not need to scroll the list, you may remove the scroll bar.

For more information about disabling scroll bar arrows, see Chapter 6, "Windows."

When incorporating a list box into a window's design, consider supporting both command (Cut, Copy and Paste) and direct manipulation (drag and drop) transfers for the list box. For example, if the list displays icons or values that the user can move or copy to other locations, such as another list box, support transfer operations for the list. The list view control automatically supports this; however, the system provides support for you to enable this for other list boxes as well.

List boxes can be classified by how they display a list and by the type of selection they support.

Single Selection List Boxes

A *single selection list box* is designed for the selection of only one item in a list. Therefore, the control provides a mutually exclusive operation similar to a group of option buttons, except that a list box can more efficiently handle a large number of items.

Define a single selection list box to be tall enough to show at least three to eight choices, as shown in Figure 7.13—depending on the design constraints of where the list box is used. Always include a vertical scroll bar. If all the items in the list are visible, then follow the window scroll bar guidelines for disabling the scroll arrows and enlarging the scroll box to fill the scroll bar shaft.

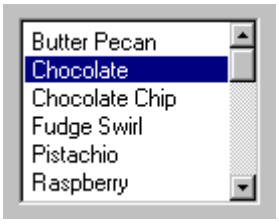


Figure 7.13 A single selection list box

The currently selected item in a single selection list box is highlighted using selection appearance.

The user can select an entry in a single selection list box by clicking on it with mouse button 1 (for pens, tapping). This also sets the input focus on that item in the list. Because this type of list box supports only single selection, when the user chooses another entry any other selected item in the list becomes unselected. The scroll bar in the list box allows the mouse user to scroll through the list of entries, following the interaction defined for scroll bars.

For more information about the interaction techniques of scroll bars, see Chapter 6, "Windows."

The keyboard interface uses navigation keys, such as the arrow keys, HOME, END, PAGE UP, and PAGE DOWN. It also uses text keys, with matches based on timing; for example, when the user presses a text key, an entry matching that character scrolls to the top of the list and becomes selected. These keys not only navigate to an entry in the list, but also select it. If no item in the list is currently selected, when the user chooses a navigation key, the first item in the list that corresponds to that key is selected. For example, if the user presses the DOWN ARROW key, the first entry in the list is selected, instead of navigating to the second item in the list.

If the choices in the list box represent values for the property of a selection, then make the current value visible and highlighted when displaying the list. If the list box reflects mixed values for a multiple selection, then no entry in the list should be selected.

Drop-down List Boxes

Like a single selection list box, a *drop-down list box* provides for the selection of a single item from a list of items; the difference is that the list is displayed upon demand. In its closed state, the control displays the current value for the control. The user opens the list to change the value. Figure 7.14 shows the drop-down list box in its closed and opened state.

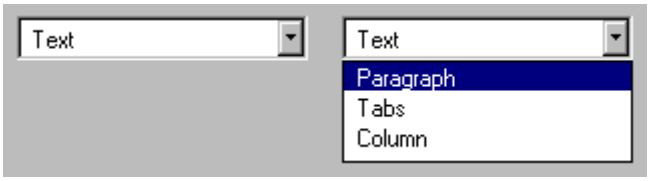


Figure 7.14 A drop-down list box (closed and opened state)

While drop-down list boxes are an effective way to conserve space and reduce clutter, they require more user interaction for browsing and selecting an item than a single selection list box.

Make the width of a closed drop-down list box a few spaces larger than the average width of the items in its list. The open list component of the control should be tall enough to show three to eight items, following the same conventions of a single selection list box. The width of the list should be wide enough not only to display the choices in the list, but also to allow the user to drag directly into the list.

The interface for drop-down list boxes is similar to that for menus. For example, the user can press the mouse button on the current setting portion of the control or on the control's menu button to display the list. Choosing an item in the list automatically closes the list.

If the user navigates to the control using an access key, the TAB key or arrow keys, an UP ARROW or DOWN ARROW, or ALT+UP ARROW or ALT+DOWN ARROW displays the list. Arrow keys or text keys navigate and select items in the list. If the user presses ALT+UP ARROW, ALT+DOWN ARROW, a navigation key, or an access key to move to another control, the list automatically closes. When the list is closed, preserve any selection made while the list was open.

If the choices in a drop-down list represent values for the property of a multiple selection and the values for that property are mixed, then display no value in the current setting component of the control.

Extended and Multiple Selection List Boxes

Although most list boxes are single selection lists, some contexts require the user to choose more than one item. *Extended selection list boxes* and *multiple selection list boxes* support this functionality.

Extended and multiple selection list boxes follow the same conventions for height and width as single selection list boxes. The height should display no less than three items and generally no more than eight, unless the size of the list varies with the size of the window. Base the width of the box on the average width of the entries in the list.

Extended selection list boxes support conventional navigation, and contiguous and disjoint selection techniques. That is, extended selection list boxes are optimized for selecting a single item or a single range, while still providing for disjoint selections.

For more information about contiguous and disjoint selection techniques, see Chapter 5, "General Interaction Techniques."

When you want to support user selection of several disjoint entries from a list, but an extended selection list box is too cumbersome, you can define a multiple selection list box. Whereas extended selection list boxes are optimized for individual item or range selection, multiple selection list boxes are optimized for independent selection.

Precede each item in a multiple selection list box with a check box, as shown in Figure 7.15. This appearance helps the user to distinguish the difference in the interface of the list box with a familiar convention. It also serves to differentiate keyboard navigation from the state of a choice. Because the check box controls are nested, you use the flat appearance style for the check boxes.

For more information about the flat appearance style for controls in a list box, see Chapter 13, "Visual Design."

Figure 7.15 A multiple selection list box

List View Controls

A *list view control* is a special extended selection list box that displays a collection of items, each item consisting of an icon and a label. List view controls can display content in four different views.

View	Description
Icon	Each item appears as a full-sized icon with a label below it. The user can drag the icons to any location within the view.
Small Icon	Each item appears as a small icon with its label to the right. The user can drag the icons to any location within the view.
List	Each item appears as a small icon with its label to the right. The icons appear in a columnar, sorted layout.
Report	Each item appears as a line in a multicolumn format with the leftmost column including the icon and its label. The subsequent columns contain information supplied by the application displaying the list view control.

The control also supports options for alignment of icons, selection of icons, sorting of icons, and editing of the icon's labels. It also supports drag and drop interaction.

Use this control where the representation of objects as icons is appropriate. In addition, provide pop-up menus on the icons displayed in the views. This provides a consistent paradigm for how the user interacts with icons elsewhere in the Windows interface.

Selection and navigation in this control work similarly to that in folder windows. For example, clicking on an icon selects it. After selecting the icon, the user can use extended selection techniques, including region selection, for contiguous or disjoint selections. Arrow keys and text keys (time-out based matching) support keyboard navigation and selection.

Tree View Controls

A *tree view control* is a special list box control that displays a set of objects as an indented outline based on their logical hierarchical relationship. The control includes buttons that allow the outline to be expanded and collapsed, as

shown in Figure 7.16. You can use a tree view control to display the relationship between a set of containers or other hierarchical elements.

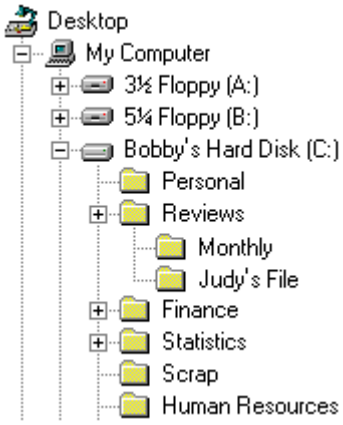


Figure 7.16 A tree view control

You can optionally include images with the text label of each item in the tree. Different images can be displayed when the user expands or collapses the item to reflect different state information.

The control supports drawing lines that define the hierarchical relationship of the items in the list and buttons for expanding and collapsing the outline. It is best to include these features (even though they are optional) because they make it easier for the user to interpret the outline.

Arrow keys provide keyboard support for navigation through the control; the user presses UP ARROW and DOWN ARROW to move between items and LEFT ARROW and RIGHT ARROW to move along a particular branch of the outline. Pressing RIGHT ARROW can also expand the outline at a branch if it is not currently displayed. Text keys can also be used to navigate and select items in the list, using the matching technique based on timing.

When you use this control in a dialog box, if you press the ENTER key or use double-clicking to carry out the default command for an item in the list, make certain that the default command button in your dialog box matches. For example, if you use double-clicking an entry in the outline to display the item's properties, then define a Properties button to be the default command button in the dialog box when the tree view control has the input focus.

Text Fields

Windows includes a number of controls that facilitate the display, entry, or editing of a text value. Some of these controls combine a basic text entry field with other types of controls.

Text fields do not include labels as a part of the control. However, you can add one using a static text field. Including a label helps identify the purpose of a text field and provides a means of indicating when the field is disabled. Use sentence capitalization for multiple word labels. You can also define access keys for the text label to provide keyboard access to the text field. When using a static text label, define keyboard access to move the input focus to the text field with which the label is associated rather than the static text field itself. You can also support keyboard navigation to text fields by using the TAB key (and, optionally, arrow keys).

For more information about static text fields, see the section, "Static Text Fields," later in this chapter. For more information about validation of input, see Chapter 8, "Secondary Windows." For more information about the visual presentation of read-only text fields, see Chapter 13, "Visual Design."

When using a text field for input of a restricted set of possible values, for example, a field where only numbers are appropriate, validate user input immediately, either by ignoring inappropriate characters or by providing feedback indicating that the value is invalid or both.

You can use text fields to display information that is read-only; that is, text that can be displayed, but not directly edited by a user. However, you can support selection of text in a read-only text field.

Text Boxes

A *text box* (also referred to as an edit control) is a rectangular control where the user enters or edits text, as shown in Figure 7.17. It can be defined to support a single line or multiple lines of text. The outline border of the control is optional, although the border is typically displayed in a toolbar or a secondary window.

Figure 7.17 A standard text box

The standard text box control provides basic text input and editing support. Editing includes the insertion or deletion of characters and the option of text wrapping. Although individual font or paragraph properties are not supported, the entire control can support a specific font setting.

A text box supports standard interactive techniques for navigation and contiguous selection. Horizontal scrolling is available for single line text boxes, and horizontal and vertical scroll bars are supported for multiple line text boxes.

You can limit the number of characters accepted as input for a text box to whatever is appropriate for the context. In addition, text boxes defined for fixed-length input can also support *auto-exit*; that is, as soon as the last character is typed in the text box, the focus moves to the next control. For example, you can define a five-character auto-exit text box to facilitate the entry of zip code, or three two-character auto-exit text boxes to support the entry of a date. Use auto-exit text boxes sparingly; the automatic shift of focus can surprise the user. They are best limited to situations involving extensive data entry.

Rich-Text Boxes

A *rich-text box*, as shown in Figure 7.18, provides the same basic text editing support as a standard text box. In addition, a rich-text box supports font properties, such as typeface, size, color, bold, and italic format, for each character and paragraph format properties, such as alignment, tabs, indents, and numbering. The control also supports printing of its content and embedding of OLE objects.

For more information about OLE embedded and linked objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

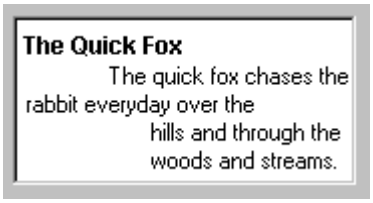


Figure 7.18 A rich-text box

Combo Boxes

A *combo box* is a control that combines a text box with a list box, as shown in Figure 7.19. This allows the user to type in an entry or choose one from the list.

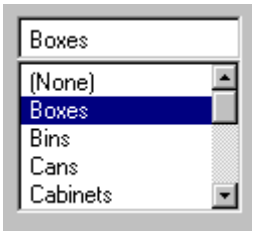


Figure 7.19 A combo box

The text box and its associated list box have a dependent relationship. As text is typed into the text box, the list scrolls to the nearest match. In addition, when the user selects an item in the list box, it automatically uses that entry to replace the content of the text box and selects the text.

The interface for the control follows the conventions supported for each component, except that the UP ARROW and DOWN ARROW keys move only in the list box.

Drop-down Combo Boxes

A *drop-down combo box*, as shown in Figure 7.20, combines the characteristics of a text box with a drop-down list box. A drop-down combo box is more compact than a regular combo box; it can be used to conserve space. The tradeoff to conserve the space is the additional user interaction required to display the list.

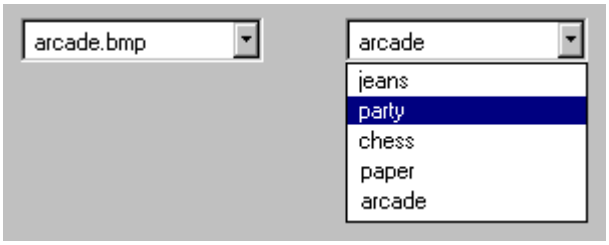


Figure 7.20 A drop-down combo box (closed and opened state)

The closed state of a drop-down combo box is similar to that of a drop-down list, except that the text box is interactive. When the user clicks the control's menu button the list is opened. Clicking the menu button a second time, choosing an item in the list, or clicking another control closes the list. Pressing the access key or shortcut key for the control navigates to the control. You can also support the TAB key or arrow keys for navigation to the control. When the control has the input focus, when the user presses the UP ARROW or DOWN ARROW or ALT+UP ARROW or ALT+DOWN ARROW key, the list is displayed.

When the control has the input focus, pressing a navigation key, such as the TAB key, or an access key or ALT+UP ARROW or ALT+DOWN ARROW to navigate to another control closes the list. When the list is closed, preserve any selection made while the list was open, unless the user presses a Cancel command button.

When the list is displayed, the interdependent relationship between the text box and list is the same as it is for standard combo boxes when the user types text into the text box. When the user chooses an item in the list, the interaction is the same as for drop-down lists; the selected item becomes the entry in the text box.

Spin Boxes

Spin boxes are text boxes that accept a limited set of discrete ordered input values that make up a circular loop. A spin box is a combination of a text box and a special control that incorporates a pair of buttons (also known as an up-down control), as shown in Figure 7.21.

Figure 7.21 A spin box

When the user clicks on the text box or the buttons, the input focus is set to the text box component of the control. The user can type a text value directly into the control or use the buttons to increment or decrement the value. The unit of change depends on what you define the control to represent.

Use caution when using the control in situations where the meaning of the buttons may be ambiguous. For example, with numeric values, such as dates, it may not be clear whether the top button increments the date or changes to the previous date. Define the top button to increase the value by one unit and the bottom button to decrease the value by one unit. Typically, wrap around at either end of the set of values. You may need to provide some additional information to communicate how the buttons apply.

By including a static text field as a label for the spin box and defining an associated access key, you can provide direct keyboard access to the control. You can also support keyboard access using the TAB key (or, optionally, arrow keys). Once the control has the input focus, the user can change the value by pressing UP ARROW or DOWN ARROW.

You can also use a single set of spin box buttons to edit a sequence of related text boxes, for example, time as expressed in hours, minutes, and seconds. The buttons affect only the text box that currently has the input focus.

Static Text Fields

You can use static text fields to present read-only text information. At the same time, your application can still alter read-only text to reflect a change in state. For example, you can use static text to display the current directory path or the status information such as page number, key states, or time and date. Figure 7.22 illustrates a static text field.

Figure 7.22 A static text field

You can also use static text fields to provide labels or descriptive information for other controls. Using static text fields as labels for other controls allows you to provide access-key activation for the control with which it is associated. Make certain that the input focus moves to its associated control and not to the static field.

Shortcut Key Input Controls

A *shortcut key input control* (also known as a hot key control) is a special kind of text box to support user input of a key or key combination to define a shortcut key assignment. Use it when you provide an interface for the user to customize shortcut keys supported by your application. Because shortcut keys carry out a command directly, they provide a more efficient interface for common or frequently used actions.

For more information about the use of shortcut keys, see Chapter 4, "Input Basics."

The control allows you to define invalid keys or key combinations to ensure valid user input; the control will only access valid keys. You also supply a default modifier to use when the user enters an invalid key. The control displays the valid key or key combination including any modifier keys.

When the user clicks a shortcut key input control, the input focus is set to the control. Like most text boxes, the control does not include its own label, so use a static text field to provide a label and assign an appropriate access key. You can also support the TAB key to provide keyboard access to the control.

Other General Controls

The system also provides support for controls designed to organize other controls and controls for special types of interfaces.

Group Boxes

A *group box* is a special control you can use to organize a set of controls. A group box is a rectangular frame with an optional label that surrounds a set of controls, as shown in Figure 7.23. Group boxes generally do not directly process any input. However, you can provide navigational access to items in the group using the TAB key or by assigning an access key to the group label.

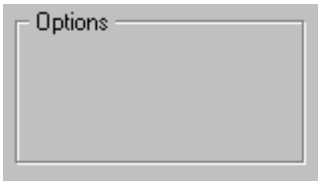


Figure 7.23 A group box

You can make the label for controls that you place in a group box relative to the group box's label. For example, a group labeled Alignment can have option buttons labeled Left, Right, and Center. Use sentence capitalization for a multiple word label.

Column Headings

Using a *column heading* control, also known as a header control, you can display a heading above columns of text or numbers. You can divide the control into two or more parts to provide headings for multiple columns, as shown in Figure 7.24. The list view control also provides support for a column heading control.

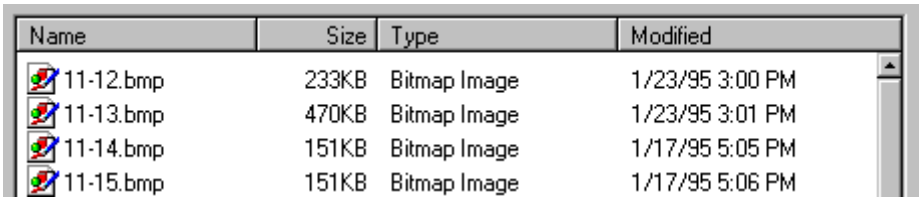


Figure 7.24 A column heading divided into four parts

You can configure each part to behave like a command button to support user tasks, such as sorting a list by clicking on a particular header part. For example, the user could sort the list displayed in Figure 7.24 by size by clicking on the Size header part.

Each header part label can include text and a graphic image. Use the graphic image to show information such as the sort direction. You can align the title elements to be left, right, or centered. The control also supports the user dragging on the divisions that separate header parts to set the width of each column.

Tabs

A *tab* control is analogous to a divider in a file cabinet or notebook, as shown in Figure 7.25. You can use this control to define multiple logical pages or sections of information within the same window.

Figure 7.25 A tab control

By default, a tab control displays only one row of tabs. If you need more tabs, the control supports multiple rows. You can also use other controls to scroll a set of tabs. But if it is important for the user to see the full range of tabs at one time, the multiple row design works better.

Tab labels can include text or graphic information, or both. Usually, the control automatically sizes the tab to the size of its label; however, you can define your tabs to have a fixed width. It is best to use the system font for the text labels of your tabs and use sentence capitalization for multiple word labels. If you use only graphics as your tab label, support tooltips for your tabs.

For more information about tooltips, see Chapter 12, "User Assistance."

When the user clicks a tab with mouse button 1, the input focus moves and switches to that tab. When a tab has the input focus, LEFT ARROW or RIGHT ARROW keys move between tabs. CTRL+TAB also switches between tabs. Optionally, you can also define access keys for navigating between tabs. If the user switches pages using the tab, you can place the input focus on the particular control on that page. If there is no appropriate control or field in which to place the tab, leave the input focus on the tab itself.

Property Sheet Controls

A *property sheet control* provides the basic framework for defining a property sheet. It provides the common controls used in a property sheet and accepts modeless dialog box layout definitions to automatically create tabbed property pages.

For more information about property sheets, see Chapter 8, "Secondary Windows."

Scroll Bars

Scroll bars are horizontal or vertical scrolling controls you can use to create scrollable areas other than on the window frame or list box where they can be automatically included. Use scroll bar controls only for supporting scrolling contexts. For contexts where you want to provide an interface for setting or adjusting values, use a slider or other control, such as a spin box. Because scroll bars are designed for scrolling information, using a scroll bar to set values inconsistently may confuse the user as to the purpose or interaction of the control.

When using scroll bar controls, follow the recommended conventions for disabling the scroll bar arrows. Disable a scroll bar arrow button when the user scrolls the information to the beginning or end of the data, unless the structure permits the user to scroll beyond the data. For more information about scroll bar conventions, see Chapter 6, "Windows."

While scroll bar controls can support the input focus, avoid defining this type of interface. Instead, define the keyboard interface of your scrollable area so that it can scroll without requiring the user to move the input focus to a

scroll bar. This makes your scrolling interface more consistent with the user interaction for window and list box scroll bars.

Sliders

Use a slider for setting or adjusting values on a continuous range of values, such as volume or brightness. A *slider* is a control, sometimes called a trackbar control, that consists of a bar that defines the extent or range of the adjustment, and an indicator that both shows the current value for the control and provides the means for changing the value, as shown in Figure 7.26.

Figure 7.26 A slider

The user can move the slide indicator by dragging to a particular location or clicking in the hot zone area of the bar, which moves the slide indicator directly to that location. To provide keyboard interaction, support the TAB key and include a static text label for access key navigation. Use sentence capitalization for a multiple word label. When the control has the input focus, arrow keys can be used to move the slide indicator in the respective direction represented by the key.

Sliders support a number of options. You can set the slider orientation as vertical or horizontal, define the length and height of the slide indicator and the slide bar component, define the increments of the slider, and whether to display tick marks for the control.

Because a slider does not include its own label, use a static text field to create one. You can also add text and graphics to the control to help the user interpret the scale and range of the control.

Progress Indicators

A *progress indicator* is a control, also known as a progress bar control, you can use to show the percentage of completion of a lengthy operation. It consists of a rectangular bar that "fills" from left to right, as shown in Figure 7.27.

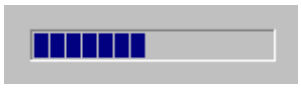


Figure 7.27 A progress indicator

Because a progress indicator only displays information, it is typically noninteractive. However, it may be useful to add static text or other information to help communicate the purpose of the progress indicator. You can use progress indicators in message boxes and status bars depending on how modal the operation or process the progress indicator represents. Use the control as feedback for long operations or background processes as a supplement to changing the pointer. The control provides more visual feedback to the user about the progress of the process. You can also use the control to reflect the progression of a background process, leaving the pointer's image to reflect interactivity for foreground activities.

For more information about message boxes, see Chapter 9, "Secondary Windows." For more information about status bars, see the section, "Toolbars and Status Bars," later in this chapter.

Tooltip Controls

A tooltip control provides the basic functionality of a tooltip. A *tooltip* is a small pop-up window that includes descriptive text displayed when the user moves the pointer over a control, as shown in Figure 7.28. The tooltip appears after a short time-out and is automatically removed when the user clicks the control or moves the pointer off the control. The tooltip is usually displayed at the lower right of the pointer, but is automatically adjusted if this location is offscreen.

For more information about the use of tooltips, see Chapter 12, "User Assistance." For more information about the use of tooltips in toolbars, see the section, "Toolbars and Status Bars," later in this chapter.

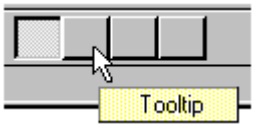


Figure 7.28 A tooltip control

Wells

A *well* is a special field similar to a group of option buttons, but facilitates user selection of graphic values such as a color, pattern, or images, as shown in Figure 7.29. This control is not currently provided by the system; however, its purpose and interaction guidelines are described here to provide a consistent interface.

Figure 7.29 A well control for selection colors

Like option buttons, use well controls for values that have two or more choices and group the choices to form a logical arrangement. When the control is interactive, use the same border pattern as a check box or text box. When the user chooses a particular value in the group, indicate the set value with a special selection border drawn around the edge of the control.

Follow the same interaction techniques as option buttons. When the user clicks a well in the group the value is set to that choice. Provide a group box or static text to label the group and define an access key for that label as well as supporting the TAB key to navigate to a group. Use arrow keys to move between values in the group

Pen-Specific Controls

When the user installs a pen input device, single line text boxes and combo boxes automatically display a writing tool button. In addition, the system provides controls for supporting pen input.

For more information about the writing tool button, see Chapter 5, "General Interaction Techniques."

Boxed Edit Controls

A *boxed edit* control provides the user with a discrete area for entering characters. It looks and operates similarly to a writing tool window without some of the writing tool window's buttons, as shown in Figure 7.30.

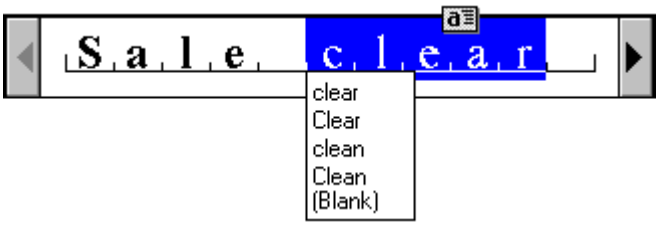


Figure 7.30 A single line boxed edit control

Both single and multiple line boxed edit controls are supported. Figure 7.31 shows a multiple line boxed edit control.

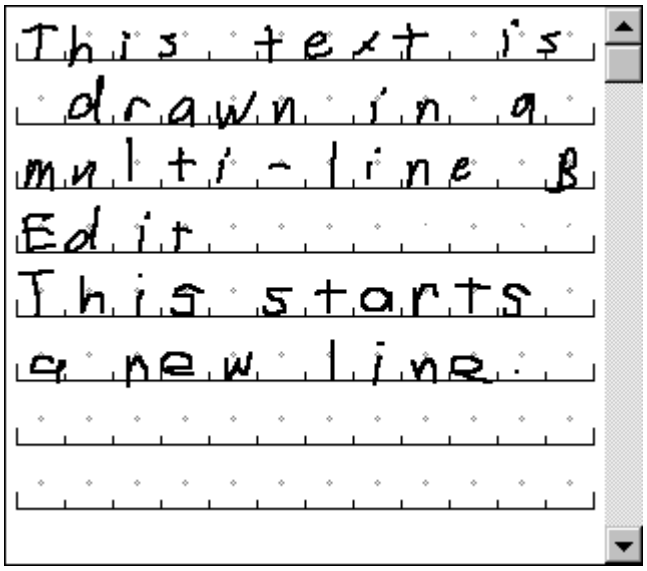


Figure 7.31 A multiple line boxed edit control

Like the writing tool window, these controls provide a pen selection handle for selection of text and an action handle for operations on a selection. They also provide easy correction by overwriting and selecting alternative choices.

Ink Edit Controls

The *ink edit* is a pen control in which the user can create and edit lines drawn as ink; no recognition occurs here. It is a drawing area designed for ink input, as shown in Figure 7.32.

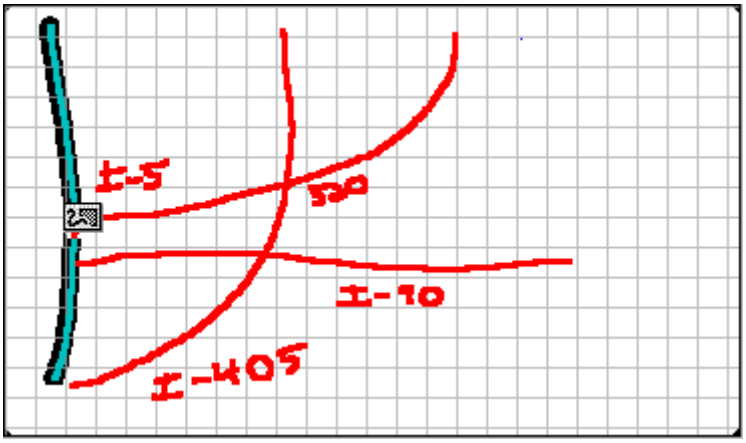


Figure 7.32 An ink edit control

The control provides support for an optional grid, optional scroll bars, and optional display of a frame border. Selection is supported using tapping to select a particular stroke; lasso-tapping is also supported for selecting single or multiple strokes. After the user makes a selection, an action handle is displayed. Tapping on the action handle displays a pop-up menu that includes commands for Undo, Cut, Copy, Paste, Delete, Use Eraser, Resize, What's This?, and Properties. Choosing the Properties command displays a property sheet associated with the selection—this allows the user to change the stroke width and color.

If you use an ink edit control, you may also want to include some controls for special functions. For example, a good addition is an Eraser button, as shown in Figure 7.33.

Figure 7.33 The eraser toolbar button

Implement the Eraser button to operate as a "spring-loaded" mode; that is, choosing the button causes the pen to act as an eraser while the user presses the pen to the screen. As soon as it is lifted, the pen reverts to its drawing mode.

Toolbars and Status Bars

Like menu bars, toolbars and status bar are special interface constructs for managing sets of controls. A *toolbar* is a panel that contains a set of controls, as shown in Figure 7.34, designed to provide quick access to specific commands or options. Specialized toolbars are sometimes called ribbons, tool boxes, and palettes.

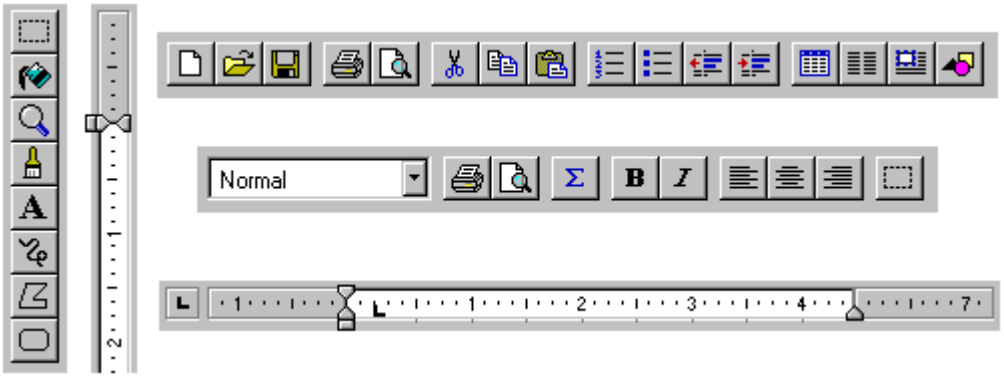


Figure 7.34 Examples of toolbars

A *status bar*, shown in Figure 7.35, is a special area within a window, typically the bottom, that displays information about the current state of what is being viewed in the window or any other contextual information, such as keyboard state. You can also use the status bar to provide descriptive messages about a selected menu or toolbar button. Like a toolbar, a status bar can contain controls; however, typically include read-only or noninteractive information.

For more information about status bar messages, see Chapter 12, "User Assistance."

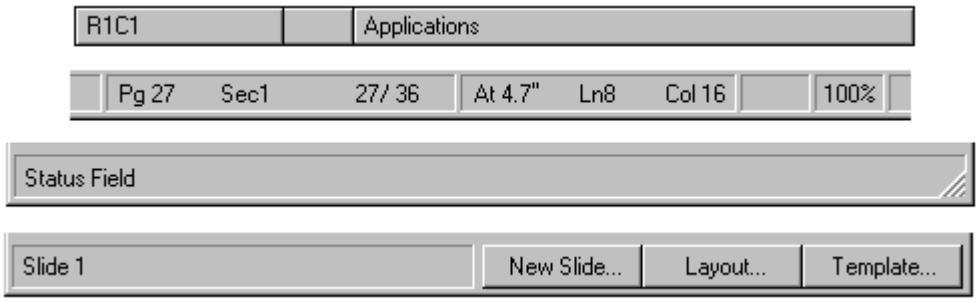


Figure 7.35 Examples of status bars

Interaction with Controls in Toolbars and Status Bars

The user can access the controls included in a toolbar or status bar with the mouse or pen through the usual means of interaction for those controls. You can provide keyboard access using either shortcut keys or access keys. If a control in a toolbar or status bar does not have a text label, access keys may not be as effective. Furthermore, if a particular access key is already in use in the primary window, it may not be available for accessing the control in the toolbar. For example, if the menu bar of the primary window is already using a particular access key, then the menu bar receives the key event.

When the user interacts with controls in a toolbar or status bar that reflect properties, any change is directly applied to the current selection. For example, if a button in a toolbar changes the property of text to bold, choosing that button immediately changes the text to bold; no further confirmation or transaction action is required. The only exception is if the control, such as a button, requires additional input from the user; then the effect may not be realized until the user provides the information for those parameters. An example of such an exception would be the selection of an object or a set of input values through a dialog box.

Always provide a tooltip for controls you include in a toolbar or status bar that do not have a text label. The system provides support for tooltips in the standard toolbar control and a tooltip control for use in other contexts.

Support for User Options

To provide maximum flexibility for users and their tasks, design your toolbars and status bars to be user configurable. Providing the user with the option to display or hide toolbars and status bars is one way to do this. You can also include options that allow the user to change or rearrange the elements included in toolbars and status bars.

Provide toolbar buttons in two sizes: 24 by 22 and 32 by 30 pixels. To fit a graphic label in these button sizes, design the images no larger than 16 by 16 and 24 by 24 pixels, respectively. In addition, support the user's the option to change between sizes by providing a property sheet for the toolbar (or status bar).

Consider also making the location of toolbars user adjustable. While toolbars are typically *docked* by default—aligned to the edge of a window or pane to which they apply—design your toolbars to be moveable so that the user can dock them along another edge or display them as a palette window.

For more information about palette windows, see Chapter 8, "Secondary Windows."

To undock a toolbar from its present location, the user must be able to click anywhere in the "blank" area of the toolbar and drag it to its new location. If the new location is within the hot zone of an edge, your application should dock the toolbar at the new edge when the user releases the mouse button. If the new location is not within the hot zone of an edge, redisplay the toolbar in a palette window. To redock the window with an edge, the user drags the window by its title bar until the pointer enters the hot zone of an edge. Return the toolbar to a docked state when the user releases the mouse button.

As the user drags the toolbar, provide visual feedback, such as a dotted outline of the toolbar. When the user moves the pointer into a hot zone of a dockable location, display the outline in its docked configuration to provide a cue to the user about what will happen when the drag operation is complete. You can also support user options such as resizing the toolbar by dragging its border or docking multiple toolbars side by side, reconfiguring their arrangement and size as necessary.

When supporting toolbar and status bar configuration options, avoid including controls whose functionality is not available elsewhere in the interface. In addition, always preserve the current position and size, and other state information, of toolbar and status bar configuration so that they can be restored to their state when the user reopens the window.

The system includes toolbar and status bar controls that you can use to implement these interfaces in your applications. The toolbar control supports docking and windowing functionality. It also supports a dialog box for allowing the user to customize the toolbar. You define whether the customization features are available to the user and what features the user can customize. The standard status bar control also includes the option of including a size grip control for sizing the window. When the status bar size grip is displayed, if the window displays a size grip at the junction of the horizontal and vertical scroll bars of a window, that grip should be hidden so that it does not appear in both locations at the same time. Similarly, if the user hides the status bar, restore the size grip at the corner of the scroll bars.

For more information about the size grip control, see Chapter 6, "Windows."

Common Toolbar Buttons

Table 7.5 illustrates the button images that you can use for common functions.

Table 7.5 Common Toolbar Buttons

16 x 16 Button	24 x 24 Button	Function
		New
		Open
		Save
		Print
		Print Preview
		Undo
		Redo
		Cut
		Copy
		Paste
		Delete
		Find
		Replace
		Properties
		Bold
		Italic

Table 7.5 Common Toolbar Buttons (*continued*)

16 x 16 Button	24 x 24 Button	Function
		Underline
		What's This? (context-sensitive Help mode)
		Show Help Topics
		Open parent folder
		View as large icons
		View as small icons
		View as list
		View as details
		Region selection tool
		Writing tool (pen)
		Eraser tool (pen)

Use these images only for the function described. Consistent use of these common tool images allows the user to transfer their learning and skills from product to product. If you use one of the standard images for a different function, you may confuse the user.

When designing your own toolbar buttons, follow the conventions supported by the standard system controls. For more information about the design of toolbar buttons, see Chapter 13, "Visual Design."

CHAPTER 8

Secondary Windows

Most primary windows require a set of secondary windows to support and supplement a user's activities in the primary windows. Secondary windows are similar to primary windows but differ in some fundamental aspects. This chapter covers the common types of secondary windows, such as property sheets, dialog boxes, palette windows, and message boxes.

Characteristics of Secondary Windows

While secondary windows share some characteristics with primary windows, they also differ from primary windows in their behavior and use. For example, secondary windows do not appear on the taskbar. Secondary windows obtain or display supplemental information which is often related to the objects that appear in a primary window.

Appearance and Behavior

A typical secondary window, as shown in Figure 8.1, includes a title bar and frame; a user can move it by dragging its title bar. However, a secondary window does not include Maximize and Minimize buttons, because these sizing operations typically do not apply to a secondary window. A Close button can be included to dismiss the window. The title text is a label that describes the purpose of the window; the content of the label depends on the use of the window. The title bar does not include icons.

Figure 8.1 A secondary window

Like a primary window, a secondary window includes a pop-up menu with commands that apply to the window. A user can access the pop-up menu for the window using the same interaction techniques as primary windows.

A secondary window can also include a What's This? button in its title bar. This button allows a user to display context-sensitive Help information about the components displayed in the window.

Interaction with Other Windows

Secondary windows that are displayed because of a command carried out within a primary window depend on the state of the primary window; that is, when the primary window is closed or minimized, its secondary windows are also closed or hidden. When the user reopens or restores the primary window, the secondary windows are restored to their former positions and states. However, if opening a secondary window is the result of an action outside of the object's primary window—for example, if the user carries out the Properties command on an icon in a folder or on the desktop—then the property sheet window is independent and appears as a peer with any primary windows, though it does not appear in the taskbar.

When the user opens or switches to a secondary window, it is activated or deactivated like any other window. With the mouse or pen, the user activates a secondary window in the same way as a primary window. With the keyboard, the ALT+F6 key combination switches between a secondary window and its primary window, or other peer secondary windows that are related to its primary window.

When the user activates a primary window, bringing it to the top of the window Z order, all of its dependent secondary windows also come to the top, maintaining their same respective order. Similarly, activating a dependent secondary window brings its primary window and related peer windows to the top.

A dependent secondary window always appears on top of its associated primary window, layered with any related window that is a peer secondary window. When activated, the secondary window appears on top of its peers. When a peer is activated, the secondary window appears on top of its primary window, but behind the newly activated secondary window that is a peer.

You can design a secondary window to appear at the top of its peer secondary windows. Typically, you should use this technique only for palette windows and, even in this situation, make this feature configurable by the user by providing an Always On Top property setting for the window. If you support this technique for multiple secondary windows, then the windows are managed in their own Z order within the collection of windows of which they are a part.

Avoid having a secondary window with this behavior appear on top of another application's primary window (or any of the other application's dependent secondary windows) when the user activates a window of that application, unless the Always On Top window can also be applied to that application's windows.

For more information about a window’s pop-up menu, see Chapter 7, “Menus, Controls, and Toolbars.”

When the user chooses a command that opens a secondary window, use the context of the operation to determine how to present that window. In property sheets, for example, set the values of the properties in that window to represent the selection. Generally, support a model of persistence of state, displaying the window in the same state as the user last accessed it. For example, the Open dialog box preserves the current directory setting between the openings of a window. Similarly, if you use tabbed pages for navigating through information in a secondary window, display the last page the user was viewing when the user closed the window. This makes it easier for the user to repeat an operation that is associated with the window. It also provides a sense of stability in the interface. However, if a command or task implies or requires that the user begin a process in a particular sequence or state, you can alternatively present a secondary window using a fixed or consistent presentation. For example, entering a record into a database may require the user to enter the data in a particular sequence. Therefore, it may be more appropriate to present the input window, displaying the first entry field.

Unfolding Secondary Windows

Typically, secondary windows cannot be resized because their purpose is to provide concise, predefined information; the exception is some forms of palette windows. There may be situations when it is appropriate to expand the size of the window to reveal additional options—a form of progressive disclosure. In this case, you can use a command button with a label followed by two "greater than" characters (>>), sometimes referred to as an *unfold button*, to indicate that there are special options. When the user chooses the button, the secondary window expands to its alternative fixed size. As an option, you can use the button to “refold” the additional part of the window.

Cascading Secondary Windows

You can also provide the user access to additional options by including a command button that opens another secondary window. If the resulting window is independent in its operation, close the secondary window from which

the user opened it and display only the new window. However, if the intent of the subsequent window is to obtain information for a field in the original secondary window, then the original should remain displayed and the dependent window should appear on top, offset slightly to the right and below the original secondary window. When using this latter method, limit the number of secondary windows to a single level to avoid creating a cluttered cascading chain of hierarchical windows.

Window Placement

The placement of a secondary window depends on a number of factors, including the use of the window, the overall display dimensions, and the reason for the appearance of the window. In general, display a secondary window where it last appeared. If the user has not yet established a location for the window, place the window in a location that is convenient for the user to navigate to and that fully displays the window. If neither of these guidelines apply, horizontally center the secondary window within the primary window, just below the title bar, menu bar, and any docked toolbars.

Modeless vs. Modal

A secondary window can be modeless or modal. A *modeless* secondary window allows the user to interact with either the secondary window or the primary window, just as the user can switch between primary windows. It is also well suited to situations where the user wants to repeat an action — for example, finding the occurrence of a word or formatting the properties of text.

A *modal* secondary window requires the user to complete interaction within the secondary window and close it before continuing with any further interaction outside the window. A secondary window can be modal in respect to its primary window or the system. In the latter case, the user must respond and close the window before interacting with any other windows or applications.

Because modal secondary windows restrict the user's choice, use them sparingly. Limit their use to situations when additional information is required to complete a command or when it is important to prevent any further interaction until satisfying a condition. Use system modal secondary windows only in severe situations — for example, if an impending fatal system error or unrecoverable condition occurs.

Default Buttons

When defining a secondary window, you can assign the ENTER key to activate a particular command button, called the *default button*, in the window. By default, the system provides a visual designation for distinguishing the default button from other command buttons with a bold outline that appears around the button.

By definition, the default button carries out the most likely action at a given time, such as a confirmation action or an action that applies transactions made in the secondary window. Avoid making a command button the default button if its action is potentially destructive. For example, in a text search and substitution window, do not use a Replace All button as the default button for the window.

You can change the default button as the user interacts with the window. For example, if the user navigates to a command button that is not the default button, the new button temporarily becomes the default. In such a case, the new default button takes on the default appearance, while the former default button loses the default appearance. Similarly, if the user moves the input focus to another control within the window that is not a command button, the original default button resumes the default.

The assignment of a default button is a common convenience for users. However, when there is no appropriate button to designate as the default button or another control requires the ENTER key (for example, entering new lines in a multiline text control), it may not be appropriate to define a default button for the window. Alternatively, when a

particular control has the input focus and requires use of the ENTER key, you can temporarily have no button defined as the default. Then when the user moves the input focus out of the control, you can restore the default button.

Optionally, you can use double-clicking on single selection control, such as an option button or single selection list, to set or select the option and carry out the default button of the secondary window.

Navigation in Secondary Windows

With the mouse and pen, navigation to a particular field or control involves the user pointing to the field and clicking or tapping it. For button controls, this action also activates that button. For example, for check boxes, it toggles the check box setting and for command buttons, it carries out the command associated with that button.

The keyboard interface for navigation in secondary windows uses the TAB and SHIFT+TAB keys to move between controls, to the next and previous control, respectively. Each control has a property that determines its place in the navigation order. Set this property such that the user can move through the secondary window following the usual conventions for reading in western countries: left-to-right and top-to-bottom. Start with the primary control the user interacts with; for countries using roman alphabets, this control is usually located in the upper left corner. Order controls such that the user can progress through the window in a logical sequence, proceeding through groups of related controls. Command buttons for handling transactions are usually at the end of the order sequence.

You need not provide TAB access to every control in the window. When using static text as a label, the control you associated it with is the appropriate navigational destination, not the static text field itself. In addition, combination controls such as combo boxes, drop-down combo boxes, and spin boxes are considered single controls for navigational purposes. Because option buttons typically appear as a group, use the TAB key for moving the input focus to the current set choice in that group, but not between individual options — use arrow keys for this purpose. For a group of check boxes, provide TAB navigation to each control because their settings are independent of each other.

Generally, you can also use arrow keys to support keyboard navigation between controls—except when controls use these keys for their internal interface. It is acceptable to use them in addition to the TAB navigation technique wherever the control is not using the keys. For example, you can use the UP ARROW and DOWN ARROW keys to navigate between single-line text boxes or within a group of check boxes or command buttons. Always use arrow keys to navigate between option button choices and within list box controls.

You can also use access keys to provide navigation controls within a secondary window. This allows the user to access a control by pressing and holding the ALT key and an alphanumeric key that matches the access key character designated in the label of the control.

When the user presses an unmodified alphanumeric key that matches a label's access key, the key navigates to the control if the keyboard is not being used as input for another control that currently has the keyboard input focus. For example, if the input focus is currently on a check box control and the user presses an alphanumeric key, the input focus moves to the control with the matching access key. However, if the input focus is in a text box or list box, an alphanumeric key is used as text input for that control so the user cannot use it for navigation within the window without modifying it with the ALT key.

Access keys not only allow the user to navigate to the matching control, they have the same effect as clicking the control with the mouse. For example, pressing the access key for a command button carries out the action associated with that button. To ensure the user direct access to all controls, select unique access keys within a secondary window.

For more information about guidelines for selecting access keys, see Chapter 4, "Input Basics."

You can use access keys to support navigation to a control, but then return the input focus to the control from which the user navigated. For example, when the user presses the access key for a specific command button that modifies the content of a list box, you can return the input focus to the list box after the command has been carried out.

OK and Cancel command buttons are typically not assigned access keys if they are the primary transaction keys for a secondary window. In this case, the ENTER and ESC keys, respectively, provide access to these buttons.

Pressing ENTER always navigates to the default command button, if one exists, and invokes the action associated with that button. If there is no current default command button, then a control can use the ENTER key for its own use.

Validation of Input

Validate the user's input for a field or control in a secondary window as closely to the point of input as possible. Ideally, input is validated when it is entered for a particular field. You can either disallow the input, or use audio and visual feedback to alert the user that the data is not appropriate. You can also display a message box, particularly if the user repeatedly tries to enter invalid input. You can also reduce invalid feedback by using controls that limit selection to a specific set of choices — for example, check boxes, option buttons, drop-down lists — or preset the field with a reasonable default value.

If it is not possible to validate input at the point of entry, consider validating the input when the user navigates away from the control. If this is not feasible, then validate it when the transaction is committed, or whenever the user attempts to close the window. At that time, leave the window open and display a message; after the user dismisses the message, set the input focus to the control with the inappropriate data.

Property Sheets and Inspectors

You can display the properties of an object in the interface in a number of ways. For example, some folder views display the file storage properties of an object. The image and name of an icon on the desktop also reflect certain properties of that object. Other interface conventions, such as toolbars, status bars, or even scroll bars, can reflect certain properties. The most common presentation of an object's properties is a secondary window, called a property sheet. A *property sheet* is a modeless secondary window that displays the user-accessible properties of an object—that is, viewable and often editable properties. Display a property sheet when the user chooses the Properties command for an object.

Property Sheet Interface

The title bar text of the property sheet identifies the displayed object. If the object has a name, use its name and the word "Properties". If the combination of the name plus "Properties" exceeds the width of the title bar, the system truncates the name and adds an ellipsis. If the object has no name, use the object's type name. If the property sheet represents several objects, then use the objects' short type name. Where the type name cannot be applied—for example, because the selection includes heterogeneous types—substitute the word "Selection" for the short type name.

Because there can be numerous properties for an object and its context, categorize and group properties as sets within the property window. There are two techniques for supporting navigation to groups of properties in a property sheet. The first is a tabbed *property page*. Each set of properties is presented within the window as a page with a tab labeled with the name of the set. Use tabbed property pages for grouping peer-related property sets, as shown in Figure 8.2.

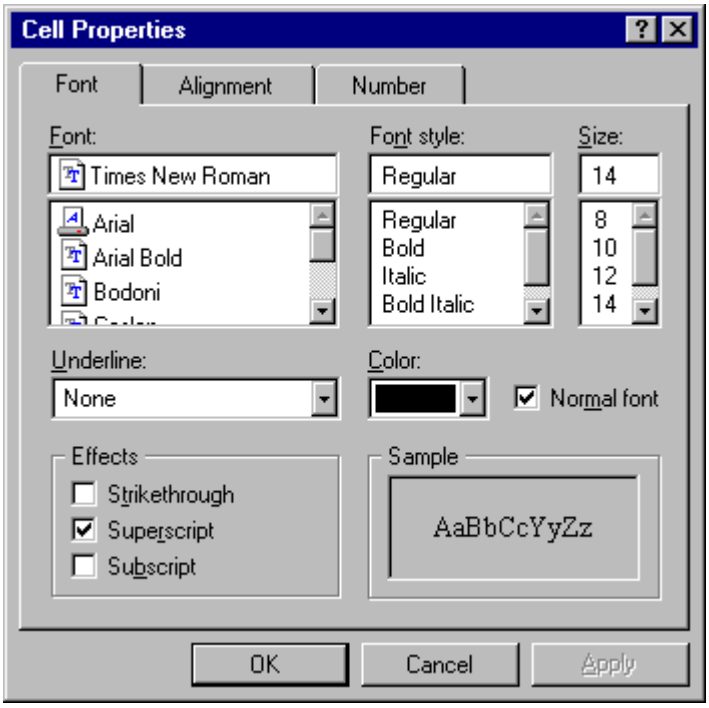


Figure 8.2 A property sheet with tabbed pages

When displaying the property sheet of an object, you provide access to the properties of the object's immediate context or hierarchically related properties in the property sheet. For example, if the user selects text, you may want to provide access to the properties of the paragraph of that text in the same property sheet. Similarly, if the user selects a cell in a spreadsheet, you may want to provide access to its related row and column properties in the same property sheet. While you can support this with additional tabbed pages, better access may be facilitated using another control—such as a drop-down list—to switch between groups of tabbed pages, as shown in Figure 8.3.

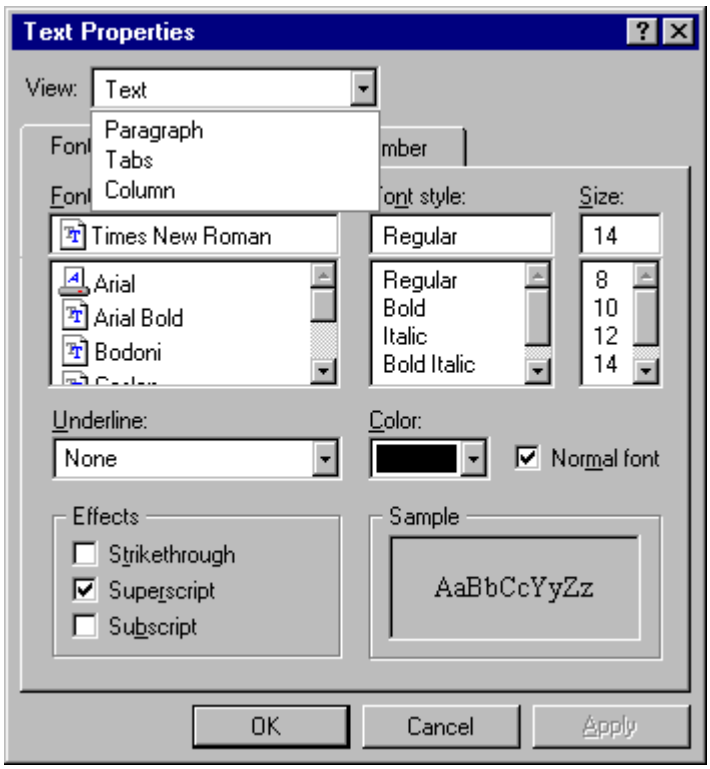


Figure 8.3 A drop-down list for access to hierarchical property sets

Where possible, make the values for properties found in property sheets transferable. You can support special transfer completion commands to enable copying only the properties of an object to another object. For example, you may want to support transferring data for text boxes or items in a list box.

For more details on transfer operations, see Chapter 5, “General Interaction Techniques."

Property Sheet Commands

Property sheets typically allow the user to change the values for a property and then apply those transactions. Include the following common command buttons for handling the application of property changes.

Command	Action
OK	Applies all pending changes and closes the property sheet window.
Apply	Applies all pending changes but leaves the property sheet window open.
Cancel	Discards any pending changes and closes the property sheet window. Does not cancel or undo changes that have already been applied.

Note Optionally, you can also support a Reset command for canceling pending changes without

closing the window.

You can also include other command buttons in property sheets. However, the location of command buttons within the property sheet window is very important. If you place a button on the property page, then apply the action associated with the button to that page. For command buttons placed outside the page but still inside the window, apply the command to the entire window.

For the common property sheet transaction buttons—OK, Cancel, and Apply—it is best to place the buttons outside the pages because users consider the pages to be just a simple grouping or navigation technique. This means that if the user makes a change on one page, the change is not applied when the user switches pages. However, if the user makes a change on the new page and then chooses the OK or Apply command buttons, both changes are applied—or, in the case of Cancel, discarded.

If your design requires groups of properties to be applied on a page-by-page basis, then place OK, Cancel, and Apply command buttons on the property pages, always in the same location on each page. When the user switches pages, any property value changes for that page are applied, or, you can prompt the user with a message box whether to apply or discard the changes.

You can include a sample on a property page to illustrate a property value change that affects the object when the user applies the property sheet. Where possible, include the aspect of the object that will be affected in the sample. For example, if the user selects text and displays the property sheet for the text, include part of the text selection in the property sheets sample. If displaying the actual object—or a portion of it—in the sample is not practical, use an illustration that represents the object's type.

Closing a Property Sheet

If the user closes a property sheet window, follow the same convention as closing the content view of an object, such as a document. Avoid interpreting the Close button as Cancel. If there are pending changes that have not been committed, prompt the user to apply or discard the changes through a message box, as shown in Figure 8.4. If there are no unsaved changes, just close the window.

Figure 8.4 Prompting for pending property changes

If the user chooses the Yes button, the properties are applied and the message box window and the property sheet window are removed. If the user chooses the No button, the pending changes are discarded and the message box and property sheet windows are closed. Include a Cancel button in the message box, to allow the user to cancel the closing of the property sheet window.

Property Inspectors

A *property inspector* is different from a property sheet—even when a property sheet window is modeless, the window is typically modal with respect to the object for which it displays properties. If the user selects another

object, the property sheet continues to display the properties of the original object. You can also display properties of an object using a dynamic viewer or browser that reflects the properties of the current selection. Such a property window is called a property inspector. When designing a property inspector, use a toolbar or palette window, or preferably a toolbar that the user can configure as a docked toolbar or palette window, as shown in Figure 8.5.

For more information about supporting docked and windowed toolbars, see Chapter 7, "Menus, Controls, and Toolbars." For more information about palette windows, see the section, "Palette Windows," later in this chapter.

Figure 8.5 A property inspector

Property transactions that the user makes in a property inspector are applied dynamically. That is, a property value is changed in the selected object as soon as the user makes the change in the control reflecting that property value.

Property inspectors and property sheets are not exclusive interfaces; you can include both. Each has its advantages. You can choose to display only the most common or frequently accessed properties in a property inspector and the complete set in the property sheet. You also can include multiple property inspectors, each optimized for managing certain types of objects.

You also can provide an interface for the user to change the behavior between a property sheet and a property inspector form of interaction. For example, you can provide a control on a property inspector that "locks" its view to be modal to the current object rather than tracking the selection.

Properties of a Multiple Selection

When a user selects multiple objects and requests the properties for the selection, reflect the properties of all the objects in a single property sheet rather than opening multiple property sheet windows. Where the property values differ, display the controls associated with those values using the mixed value appearance—sometimes referred to as the indeterminate state. Also support the display of multiple property sheets when the user displays the property sheet of the objects individually. This convention provides the user with sufficient flexibility and control over window clutter. If your design still requires access to individual properties when the user displays the property sheet of a multiple selection, include a control such as a list box or drop-down list in the property window for switching between the properties of the objects in the set.

For more information about mixed value appearance, see Chapter 13, “Visual Design.”

Properties of a Heterogeneous Selection

When a multiple selection includes different types of objects, include the intersection of the properties between the objects in the resulting property sheet. If the container of those selected objects treats the objects as if they were of a single type, the property sheet includes properties for that type only. For example, if the user selects text and an embedded object, such as a circle, and in that context an embedded object is treated as an element within the text stream, present only the text properties in the resulting property sheet.

Properties of Grouped Items

Do not equate a multiple selection with a grouped set of objects. A group is a stronger relationship than a simple selection, because the aggregate resulting from the grouping can itself be considered an object, potentially with its

own properties and operations. Therefore, if the user requests the properties of a grouped set of items, display the properties of the group or composite object. The properties of its individual members may or may not be included, depending on what is most appropriate.

Dialog Boxes

A *dialog box* provides an exchange of information or dialog between the user and the application. Use a dialog box to obtain additional information from the user—information needed to carry out a particular command or task.

Because dialog boxes generally appear after choosing a particular menu item (including pop-up or cascading menu items) or a command button, define the title text to be the name of the associated command for the window. Do not include an ellipsis and avoid including the command's menu title unless necessary to compose a reasonable title for the dialog box. For example, for a Print command on the File menu, define the dialog box window's title text as Print, not Print... or File Print. However, for an Object... command on an Insert menu, title the dialog box as Insert Object.

Dialog Box Commands

Like property sheets, dialog boxes commonly include OK and Cancel command buttons. Use OK to apply the values in the dialog box and close the window. If the user chooses Cancel, the changes are ignored and the window is closed, canceling the operation the user chose. OK and Cancel buttons work best for dialog boxes that allow the user to set the parameters for a particular command. Typically, define OK to be the default command button when the dialog box window opens.

You can include other command buttons in a dialog box in addition to or replacing the OK and Cancel buttons. Label your command buttons to clearly define the button's purpose, but be as concise as possible. Long, wordy labels make it difficult for the user to easily scan and interpret a dialog box's purpose. Follow the design conventions for command buttons.

For more information about command buttons, see Chapter 7, "Menus, Controls, and Toolbars," and Chapter 13, "Visual Design."

You can also provide a Help command button to provide Help information about the dialog box. This button provides a different form of user assistance than the title bar context-sensitive Help button.

Layout

Orient controls in dialog boxes in the direction people read. In countries where roman alphabets are used, this means left to right, top to bottom. Locate the primary field with which the user interacts as close to the upper left corner as possible. Follow similar guide lines for orienting controls within a group in the dialog box.

Lay out the major command buttons either stacked along the upper right border of the dialog box or lined up across the bottom of the dialog box. Position the most important button — typically the default command — as the first button in the set. If you use the OK and Cancel buttons, group them together. If you include a Help command button, make it the last button in the set. You can use other arrangements if there is a compelling reason, such as a natural mapping relationship. For example, it makes sense to place buttons labeled North, South, East, and West in a compass-like layout. Similarly, a command button that modifies or provides direct support for another control may be grouped or placed next to those controls. However, avoid making that button the default button because the user will expect the default button to be in the conventional location.

For more information about layout and spacing, see Chapter 13, “Visual Design.” For more information about Help command buttons, see Chapter 12, "User Assistance."

Common Dialog Box Interfaces

The system provides prebuilt interfaces for many common operations. Use these interfaces where appropriate. They can save you time while providing a high degree of consistency. If you customize or provide your own interfaces, maintain consistency with the basic functionality supported in these applications. For example, if you provide your own property sheet for font properties, model your design to be similar in appearance and design to the common font dialog box. Consistent visual and operational styles will allow users to more easily transfer their knowledge and skills.

> **Note** These common dialog box interfaces have been revised from the ones provided in previous releases of Microsoft Windows.

In all these dialog boxes, preserve the user's latest changes to the settings for the controls for subsequent openings of the dialog box while the application is running.

Open Dialog Box

The Open dialog box, as shown in Figure 8.6, allows the user to browse the file system, including direct browsing of the network, and includes controls to open a specified file. Use this dialog box to open files or browse for a filename, such as the File Open menu command or a Browse command button. Always set the title text to correctly reflect the command that displays the dialog box.

Figure 8.6 The Open dialog box

The dialog box automatically handles the display of long filenames, direct manipulation transfers, such as drag and drop, and access to an icon’s pop-up menus. The dialog box only displays filename extensions for files of registered types when the user selects this viewing option.

Even when the dialog box does not display the extensions of filename, it still supports the opening or filtering of files displayed based on the user input of an extension. For example, if the user types in *.TXT and clicks the Open button, the list displays only files with the type extension of .TXT. When this occurs, display the respective type as the setting for the Files of Type drop-down list box. If the application does not support that type, display the Files of Type control with the mixed-case appearance—that is, no setting.

For more information about the mixed-case appearance, see Chapter 13, "Visual Design."

Selecting an item in the Look In drop-down list box or opening an directory in the files list box allows the user to change the path, displaying the contents of that path location using the type setting as a filter. If the user changes the current path setting in either the Open or Save As dialog box, preserve that setting for subsequent openings of the Open dialog box for that application's open session while the application is running. Note that, changing the path in the Open dialog box only changes the path in the Save As dialog box if the file has not yet been saved.

Use the Files of the Type drop-down list box control to list the file types your application can support. Use the registered short type names for list items. Make the default type the most common extension used with your application.

The Open dialog box also displays shortcut icons to files matching the setting of the Files of Type control. When the user opens a shortcut icon, the dialog box opens the file of the object to which the link refers. In other words, the effect is the same as if the user directly opened the original file. Therefore, display the name of the original file in the window's title bar, not the name of the file link. Similarly, when saving the file, save the changes back to the original file.

Save As Dialog Box

The Save As dialog box, shown in Figure 8.7, is designed to save a file using a particular name, location, and format. Display this dialog box when the user chooses the Save command and a filename has not been supplied or confirmed by the user. This same dialog box is the one the user chooses with the Save As command, but with the title bar text changed to Save As. If you use this dialog box for other tasks that require saving files, define the title text of the dialog box to appropriately reflect that command.

Figure 8.7 The Save As dialog box

This appearance and operation of the Save As dialog box is similar to the Open dialog box except that the Files of Type field also defines how the file will be saved. Supply the file types supported by your application following the same conventions as the Open dialog box. Do not include file formats. While a format can be related to its type, a format and a type are not the same thing. For example, a bitmap file can be stored in monochrome, 16-, 256-, or 24-bit color format, but the file's type for all of them is the same. If you need to support user selection of a format, add a control to the dialog box for this purpose.

Include a default name in the File Name text box. Use the current name of the file. If the file has not been named yet, define a name based on the registered short type name for the file. If the file already exists, save the file back to its original location. If the file has never been saved, save the file to your application's default path setting for files or to the location defined by the user, either by typing in the path or by using the controls in the dialog box.

For more information about filenames and short type names, see Chapter 10, "Integrating with the System."

When the user supplies a name of the file, ignore any illegal characters. In addition, do not interpret a period as a designation for defining an extension. For example, if the user types in "My Favorite Document.TXT", that file is saved as My Favorite Document.TXT and its file type does not change.

You can customize the Open and Save As dialog boxes by adding commands to the dialog box, but you cannot reconfigure the dialog box layout.

Find and Replace Dialog Boxes

The Find and Replace dialog boxes provide controls that search for a text string specified by the user and optionally replace it with a second text string specified by the user. These dialog boxes are shown in Figure 8.8.

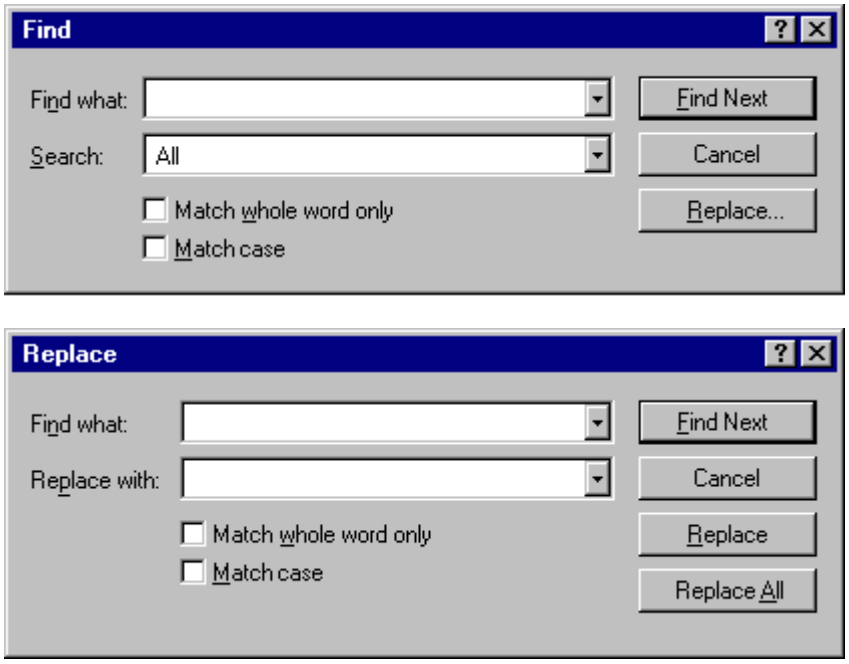


Figure 8.8 The Find and Replace dialog boxes

Print Dialog Box

The Print dialog box, shown in Figure 8.9, allows the user to select what to print, the number of copies to print, and the collation sequence for printing. It also allows the user to choose a printer and provides a command button that provides shortcut access to that printer's properties.

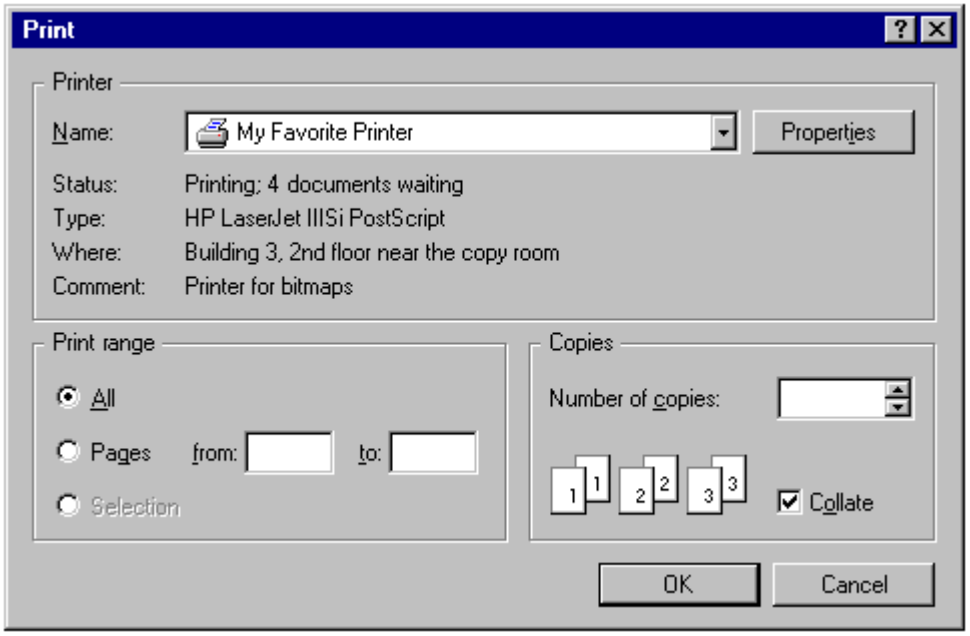


Figure 8.9 The Print dialog box

Print Setup Dialog Box

The Print Setup dialog box displays the list of available printers and provides controls for selecting a printer and setting paper orientation, size, source, and other printer properties.

> **Note** The system includes this dialog box for compatibility with applications designed for Windows 3.1. Printer properties are now provided through the property sheet of each printer, so do not include this dialog box if you are creating or updating your application to the recommendations in this guide.

Page Setup Dialog Box

The Page Setup dialog box, as shown in Figure 8.10, provides controls for specifying properties about the page elements and layout.

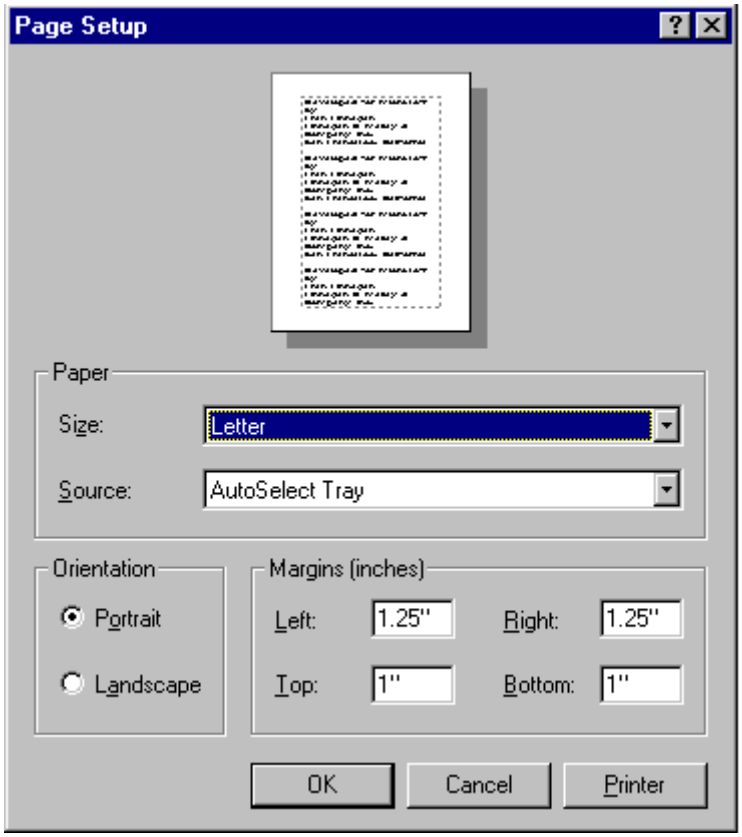


Figure 8.10 Page Setup interface used as a dialog box

In this context, page orientation refers to the orientation of the page and not the printer, which may also have these properties. Generally, the page's properties override those set by the printer, but only for the printing of that page or document.

The Printer button in the dialog box displays a supplemental dialog box (as shown in Figure 8.11) that provides information on the current default printer. Similarly to the Print dialog box, it displays the current property settings for the default printer and a button for access to the printer's property sheet.

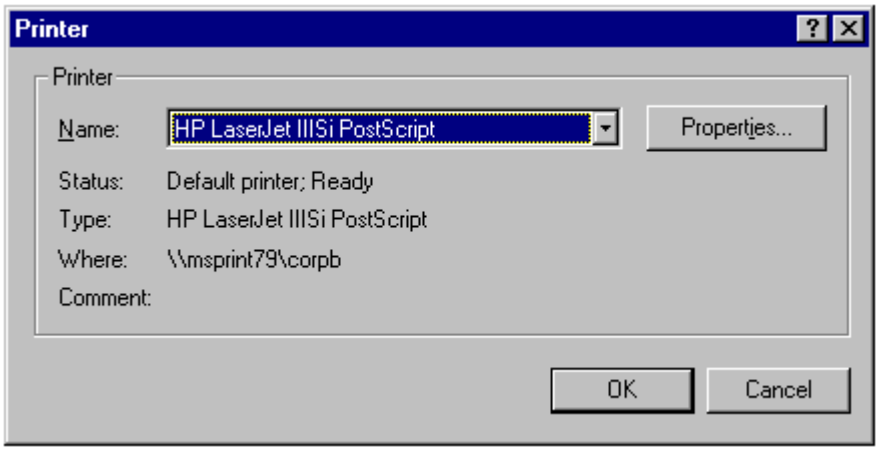


Figure 8.11 The supplemental Printer dialog box

Font Dialog Box

This dialog box displays the available fonts and point sizes of the available fonts installed in this system. You can use the Font dialog box to display or set the font properties of a selection of text. Figure 8.12 shows the Font dialog box.

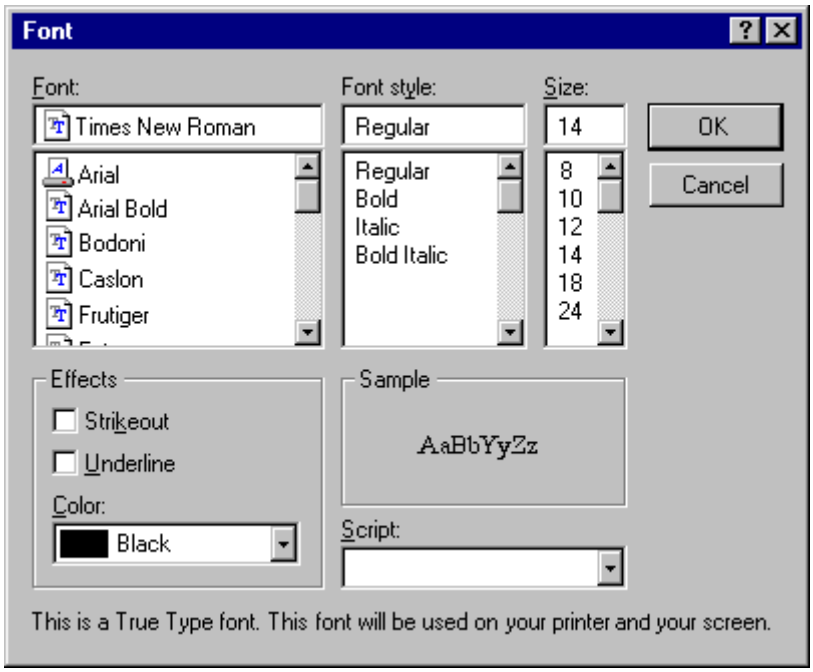


Figure 8.12 The Font dialog box

Color Dialog Box

The Color dialog box (as shown in Figure 8.13) displays the available colors and includes controls that allow the user to define custom colors. You can use this control to provide an interface for users to select colors for an object.

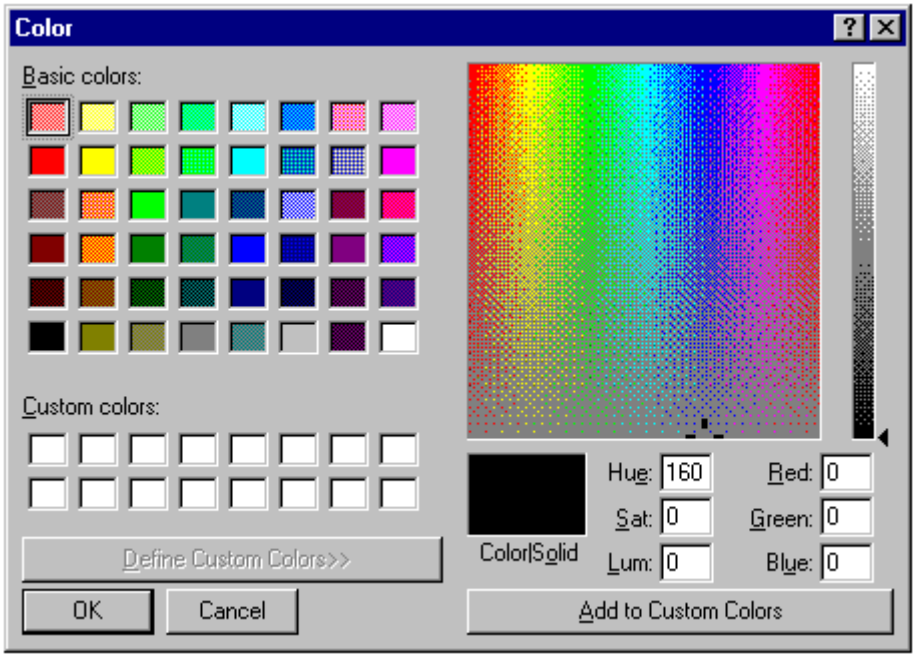


Figure 8.13 The Color dialog box

The Basic colors control displays a default set of colors. The number of colors displayed here is determined by the installed display driver. The Custom colors control allows the user to define more colors using the various color selection controls provided in the window.

Initially, you can display the dialog box as a smaller window (as shown in Figure 8.14) with only the Basic colors and Custom colors controls and allow the user to expand the dialog box to define additional colors.

Figure 8.14 The Color dialog box (unexpanded appearance)

Palette Windows

Palette windows are modeless secondary windows that present a set of controls. For example, when toolbar controls appear as a window, they appear in a palette window. Palette windows are distinguished by their visual appearance. The height of the title bar for a palette window is shorter, but it still includes only a Close button in the title area, as shown in 8.15.

Figure 8.15 A palette window

For more information about toolbars and palette windows, see Chapter 7, "Menus, Controls, and Toolbars."

Make the title text for a palette window the name of the command that displays the window or the name of the toolbar it represents, optionally followed by the word “Palette." The system supplies default size and font settings for the title bar and title bar text for palette windows.

You can define palette windows as a fixed size, or, more typically, sizable by the user. Two visual cues indicate when the window is sizable: changing the pointer image to the size pointer, and placing a Size command in the window's pop-up menu. Preserve the window's size and position so the window can be restored if it or its associated primary window is closed.

Like other windows, the title bar, the Close button, and the border areas provide an access point for the window's pop-up menu. Commands on a palette window's pop-up menu can include Close, Move, Size (if sizable), Always On Top, and Properties, as shown in Figure 8.16.

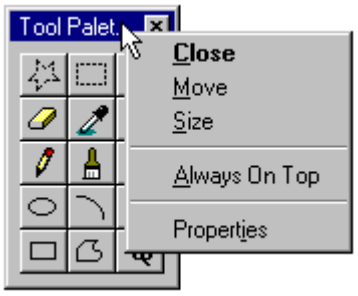


Figure 8.16 A pop-up menu for a palette window

Including the Always On Top command or property in the window's property sheet allows the user to configure the palette window to always stay at the top of the Z order of the window set of which it is a part. Setting this attribute places it at the top of other windows that are part of its related set. Turning off this option keeps the palette window within its set of related windows, but allows the user to have other windows of the set appear on top of the palette window. In this case, keep the palette window on top of the primary window of a set. This feature allows the user to configure preferred access to the palette window.

You can include a Properties command on the palette window's pop-up menu to provide an interface for allowing the user to edit properties of the window, such as the Always On Top property or a means of customizing the content of the palette window.

Message Boxes

A message box is a secondary window that displays messages; information about a particular situation or condition. Messages are an important part of the interface for any software product. Messages that are too generic or poorly written frustrate users, increase support costs, and ultimately reflect on the quality of the product. Therefore, it is worthwhile to design effective message boxes.

However, avoid creating situations that unnecessarily create a problem that requires you to display a message. For example, if there may be insufficient disk space to perform an operation, rather than assuming that you will display a message box, check before the user attempts the operation and disable the command.

Use the title bar of a message box to appropriately identify the context in which the message is displayed—usually the name of the object. For example, if the message results from editing a document, the title text is the name of that document, optionally followed by the application name. If the message results from a nondocument object, then use the application name. Providing an appropriate identifier for the message is particularly important in the Windows multitasking environment, because message boxes might not always be the result of current user interaction. In

addition, because OLE technology allows objects to be embedded, different application code may be running when the user activates the object for visual editing. Therefore, the title bar text provides an important role in communicating the nature of a message. Do not use descriptive text for message box title text such as "warning" or "caution." The message symbol conveys the nature of the message. Never use the word "error" in the title text.

You may include a message identification number as part of the message box text (not title text) for each message for support purposes. To avoid interrupting the user's ability to quickly read a message, place such a designation at the end of the message text.

Message Box Types

Each message box includes a graphical symbol that indicates what kind of message is being presented. Most messages can be classified in one of the following categories:

- Informational messages
- Warning messages
- Critical messages

The following table describes each message type and shows its associated symbol.

Table 8.1 Message Types and Associated Symbols

Symbol	Message Type	Description
	Information	Provides information about the results of a command. Offers no user choices; the user acknowledges the message by clicking the OK button.
	Warning	Alerts the user to a condition or situation that requires the user's decision and input before proceeding, such as an impending action with potentially destructive, irreversible consequences. The message can be in the form of a question—for example, "Save changes to MyReport?".
	Critical	Informs the user of a serious problem that requires intervention or correction before work can continue.

Figure 8.17 shows examples of these types of message boxes .

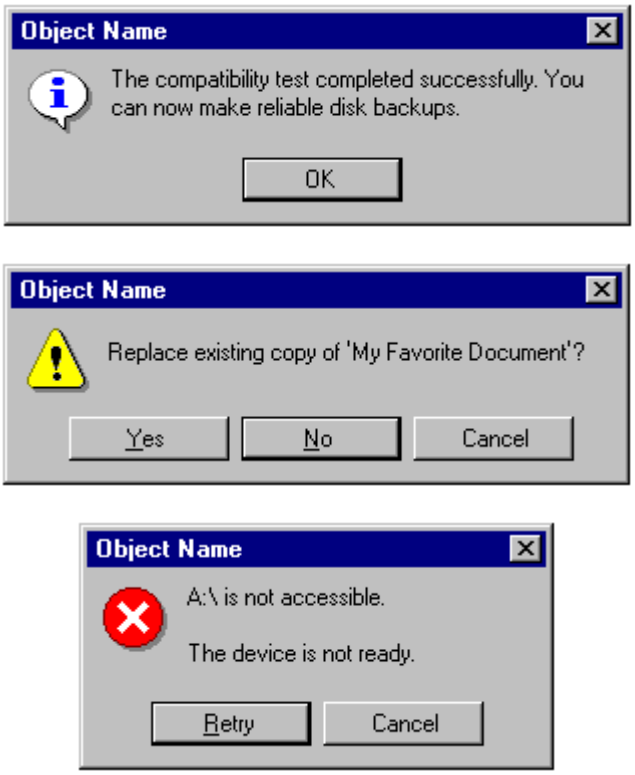


Figure 8.17 Informational, warning, and critical message boxes

Note The system does provide another message icon type that has been in earlier versions of Windows. This message icon is for cautionary messages that are phrased as a question. However, the message icon is no longer recommended as it does not clearly represent a type of message. In addition, the message can be confused with the Help symbol.

Because a message box disrupts the user's current task, it is best to display a message box only when the window of the application displaying the message box is active. If it is not active, then the application uses its entry in the taskbar to alert the user. Once the user activates the application, the message box can be displayed.

For more information about how an application uses the taskbar to alert the user, see Chapter 10, "Integrating with the System."

You can also use message boxes to provide information or status without requiring direct user interaction to dismiss them. For example, message boxes that provide a visual representation of the progress of a particular process automatically disappear when the process is complete, as shown in Figure 8.18. Similarly, product start-up windows that identify the product name and copyright information when the application starts can be automatically removed once the application has loaded. In these situations, you do not need to include a message symbol. Use this technique only for noncritical, informational messages, as some users may not be able to read the message within the short time it is displayed.

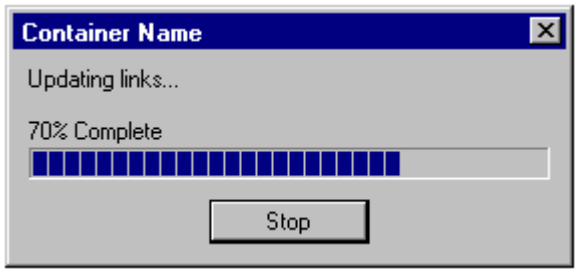


Figure 8.18 A progress message box

However, display only one message box for a specific condition. Displaying a sequential set of message boxes tends to confuse users.

Command Buttons in Message Boxes

Typically, message boxes contain only command buttons as the appropriate responses or choices offered to the user. Designate the most frequent or least destructive option as the default command button. Command buttons allow the message box interaction to be simple and efficient. If you need to add other types of controls, always consider the potential increase in complexity.

If a message requires no choices to be made but only acknowledgement, use only an OK button—and, optionally, a Help button. If the message requires the user to make a choice, include a command button for each option. The clearest way to present the choices is to state the message in the form of a question and provide a button for each response. When possible, phrase the the question to permit Yes or No answers, represented by Yes and No command buttons. If these choices are too ambiguous, label the command buttons with the names of specific actions—for example, "Save" and "Delete."

You can include command buttons in a message box that correct the action that caused the message box to be displayed. Be sure, however, to make the result of any such button's action very clear. For example, if the message box indicates that the user must switch to another application window to take corrective action, you can include a button that switches the user to that application window.

Some situations may require offering the user not only a choice between performing or not performing an action, but an opportunity to cancel the process altogether. In such situations, use a Cancel button, as shown in Figure 8.19.

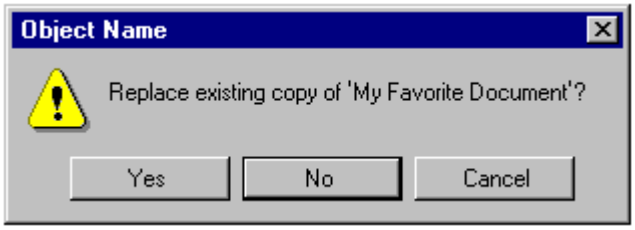


Figure 8.19 Message box choices

Note When using Cancel as a command button in a message box, remember that to users, Cancel implies restoring the state of the process or task that started the message. If you use Cancel to interrupt a process and the state cannot be restored, use Stop instead.

Message Box Text

The message text you include in a message box should be clear, concise, and in terms that the user understands. This usually means using no technical jargon or system-oriented information.

In addition, observe the following guidelines for your message text:

- State the problem, its probable cause (if possible), and what the user can do about it—no matter how obvious the solution may seem to be. For example, instead of “Insufficient disk space,” use “‘Sample Document’ could not be saved, because the disk is full. Try saving to another disk or freeing up space on this disk.”
- Consider making the solution an option offered in the message. For example, instead of "One or more of your lines are too long. The text can only be a maximum of 60 characters wide," you might say, "One or more of your lines are too long. Text can be a maximum of 60 characters in Portrait mode or 90 characters wide in Landscape. Do you want to switch to Landscape mode now?" Offer Yes and No as the choices.
- Avoid using unnecessary technical terminology and overly complex sentences. For example, "picture" can be understood in context, whereas "picture metafile" is a rather technical concept.
- Avoid phrasing that blames the user or implies user error. For example, use “Cannot find filename” instead of “Filename error.” Avoid the word “error” altogether.
- Make messages as specific as possible. No more than two or three conditions should map to a single message. For example, there may be several reasons why a file cannot be opened; provide a specific message for each condition.
- Avoid relying on default system-supplied messages, such as MS-DOS® extended error messages and Kernel INT 24 messages; instead, supply your own specific messages wherever possible.
- Be brief, but complete. Provide only as much background information as necessary. A good rule of thumb is to limit the message to two or three lines. If further explanation is necessary, provide this through a command button that opens a Help window.

Pop-up Windows

Use pop-up windows to display additional information when an abbreviated form of the information is the main presentation. For example, use a pop-up window when an entire path cannot be presented and must be abbreviated—the pop-up window displays the full path. They are often used to provide contextual Help information, as shown in Figure 8.20.

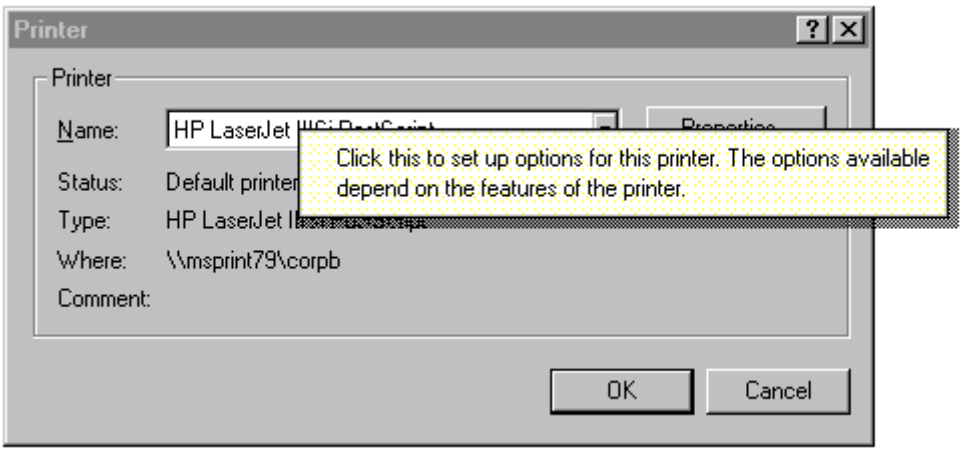


Figure 8.20 A context-sensitive Help pop-up window

Tooltips are another form of contextual Help information. They provide the name for a control in toolbars. The writing tool control is another example of the use of a pop-up window.

For more information about the use of pop-up windows for contextual Help information, see Chapter 12, "User Assistance."

How pop-up windows are displayed depends on their use, but the typical means is by the user either pointing or clicking with mouse button 1 (for pens, tapping), or an explicit command. If the user chooses the standard toolbar or tooltip controls, the system automatically provides time-outs.

If you use clicking to display a pop-up window, change the pointer as feedback to the user indicating that the pop-up window exists and requires a click. Usually, the pointing-hand cursor is the best to use. From the keyboard, use the Select key (SPACEBAR) to open and close the window.

When you use an explicit command to display a pop-up menu—for example, a command on a pop-up menu—the pop-up window is also immediately displayed. With either of these methods, clicking a second time dismisses the window.

CHAPTER 9

Window Management

User tasks can often involve working with different types of information, contained in more than one window or view. There are different techniques that you can use to manage a set of windows or views. This chapter covers some common techniques and the factors to consider for selecting a particular model.

Single Document Window Interface

In many cases, the interface of an object or application can be expressed using a single primary window with a set of supplemental secondary windows. The desktop and taskbar provide management of primary windows. Opening the window puts it at the top of the Z order and places an entry in the taskbar, making it easier for users to switch between windows without having to shuffle or reposition them. This single document window interface minimizes window clutter.

By supporting a single instance model where you activate an existing window (within the same desktop) if the user reopens the object, you make single primary windows more manageable, and reduce the potential confusion for the user. This also provides a data-centered, one-to-one relationship between an object and its window.

In addition, Microsoft OLE supports the creation of compound documents or other types of information containers. Using these constructs, the user can assemble a set of different types of objects for a specific purpose within a single primary window, eliminating the necessity of displaying or editing information in separate windows.

For more information about OLE, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Some types of objects, such as device objects, may not require a primary window and use only a secondary window for viewing and editing their properties. When this occurs, do not include the Open command in the menu for the object; instead, replace it with a Properties command, defined as the object's default command.

It is also possible for an object to have no windows; an icon is its sole representation. In this rare case, make certain that you provide an adequate set of menu commands to allow a user to control its activity.

Multiple Document Interface

For some tasks, the taskbar may not be sufficient for managing a set of related windows; for example, it can be more effective to present multiple views of the same data or multiple views of related data in windows that share interface elements. You can use the *multiple document interface* (MDI) for this kind of situation.

The MDI technique uses a single primary window, called a *parent window*, to visually contain a set of related *document* or *child windows*, as shown in Figure 9.1. Each child window is essentially a primary window, but is constrained to appear only within the parent window instead of on the desktop. The parent window also provides a visual and operational framework for its child windows. For example, child windows typically share the menu bar of the parent window and can also share other parts of the parent's interface, such as a toolbar or status bar. You can change these to reflect the commands and attributes of the active child window.

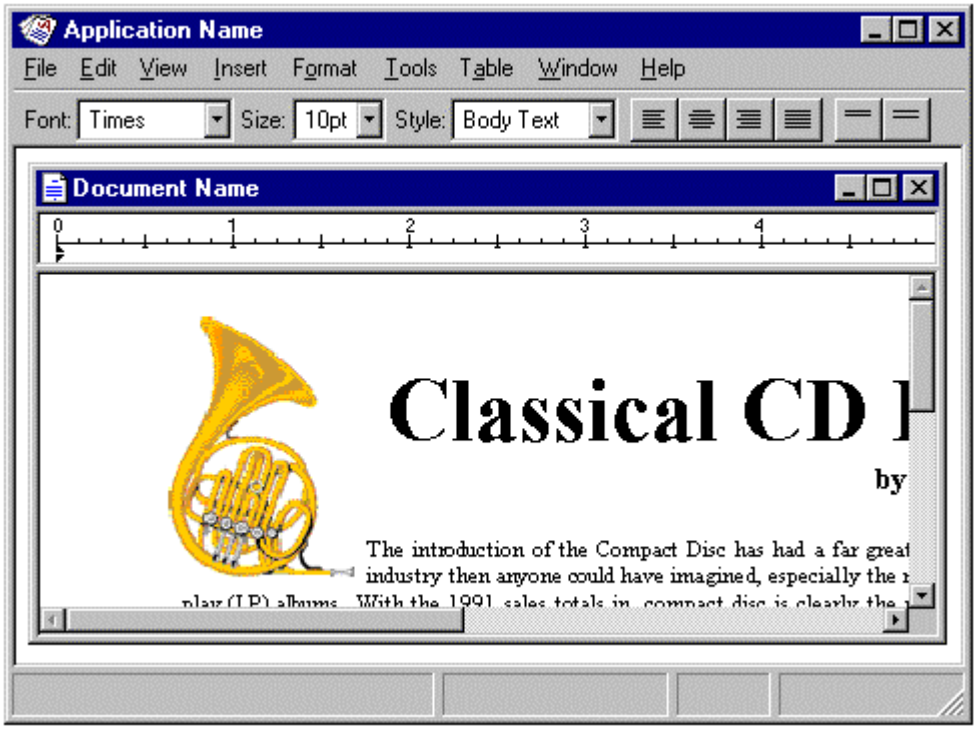


Figure 9.1 An MDI parent and child window

Secondary windows—such as dialog boxes, message boxes, or property sheets—displayed as a result of interaction within the MDI parent or child, are typically not contained or clipped by the parent window. These windows are

activated and displayed following the common conventions for secondary windows associated with a primary window, even if they apply to individual child windows.

For more information about the interaction between a primary window and its secondary windows, see Chapter 6, "Windows," and Chapter 8, "Secondary Windows."

The title bar of an MDI parent window includes the icon and name of the application or the object that represents the work area displayed in the parent window. The title bar of a child window includes the icon representing the document or data file type and its filename. The title bar also includes pop-up menus for the window and the title bar icon for both the parent window and any child windows.

Opening and Closing MDI Windows

The user starts an MDI application either by directly opening the application or by opening a document (or data file) of the type supported by the MDI application. If directly opening an MDI document, the MDI parent window opens first and then the child window for the file opens within it. To support the user opening other documents associated with the application, include an interface such as an Open command.

When the user directly opens an MDI document outside the interface of its MDI parent window — for example, by double-clicking the file — if the parent window for the application is already open, another instance of the MDI parent window is opened rather than the document's window in the existing MDI parent window. While the opening of the child window within the existing parent window can be more efficient, the opening of the new window can disrupt the task environment already set up in that parent window. For example, if the newly opened file is a macro, opening it in the opened parent window can inadvertently affect other documents open in that window. If the user wishes to open a file as part of the set in a particular parent MDI window, the commands within that window provide that support.

For more information about opening primary windows, see Chapter 6, "Windows."

Because MDI child windows are primary windows, the user can close them using the Close button in the title bar or the Close command on the pop-up menu for the window. When the user closes a child window, any unsaved changes are processed following the common conventions. Do not close its parent window, unless the parent window provides no context or operations without an open child window.

When the user closes the parent window, all of its child windows are closed. Where possible, the state of a child window is preserved, such as its size and position within the parent window; the state is restored when the file is reopened.

Moving and Sizing MDI Windows

MDI allows the user to move or hide the child windows as a set by moving or minimizing the parent window. When the user moves an MDI parent window, the open child windows within it maintain their relative positions within the parent window. Moving a child window constrains it to its parent window; in some cases, the size of the parent window's interior area may result in clipping a child window. Optionally, you can support automatic resizing of the parent window when the user moves or resizes a child window either toward or away from the edge of the parent window.

Although an MDI parent window minimizes as an entry on the taskbar, MDI child windows minimize within their parent window, as shown in Figure 9.2.

> **Note** The recommended visual appearance of a minimized child window in Microsoft Windows is now that of a window that has been sized down to display only part of its title area and its border. This avoids potential confusion between minimized child window icons and icons that represent objects.

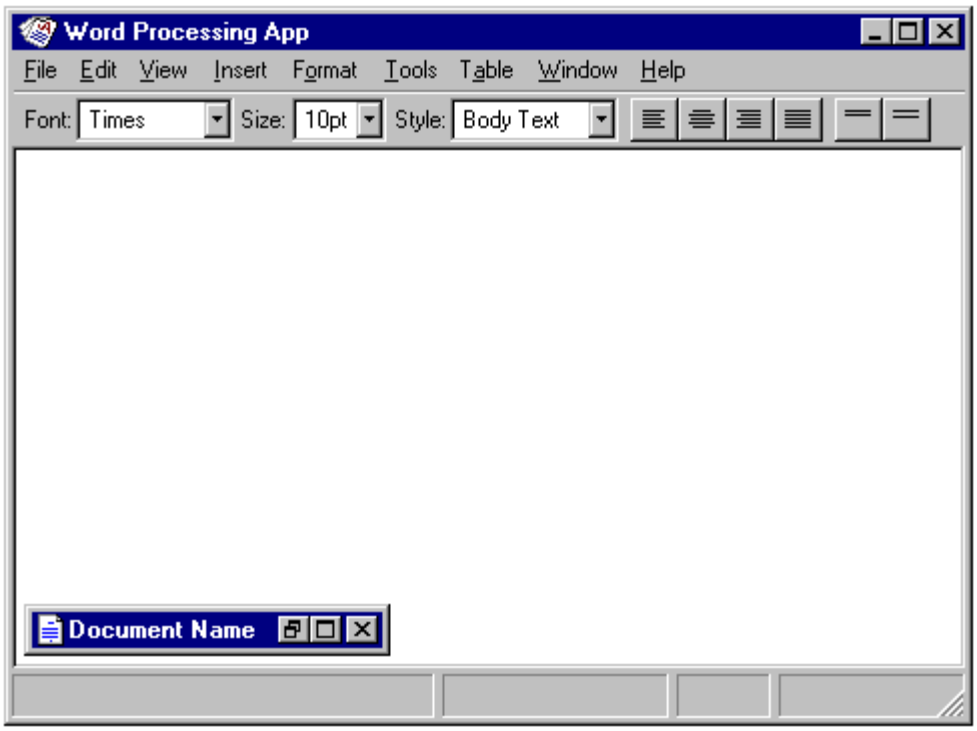


Figure 9.2 A minimized MDI child window

When the user maximizes an MDI parent window, the window expands to its maximum size, like any other primary window. When the user maximizes an MDI child window, it should also expand to its maximum size. When this size exceeds the interior of its parent window, the child window merges with its parent window. The child window's title bar icon, Restore button, Close button, and Minimize button (if supported) are placed in the menu bar of the parent window in the same relative position as in the title bar of the child window, as shown in Figure 9.3. The child window title text appends to the parent window's title text.

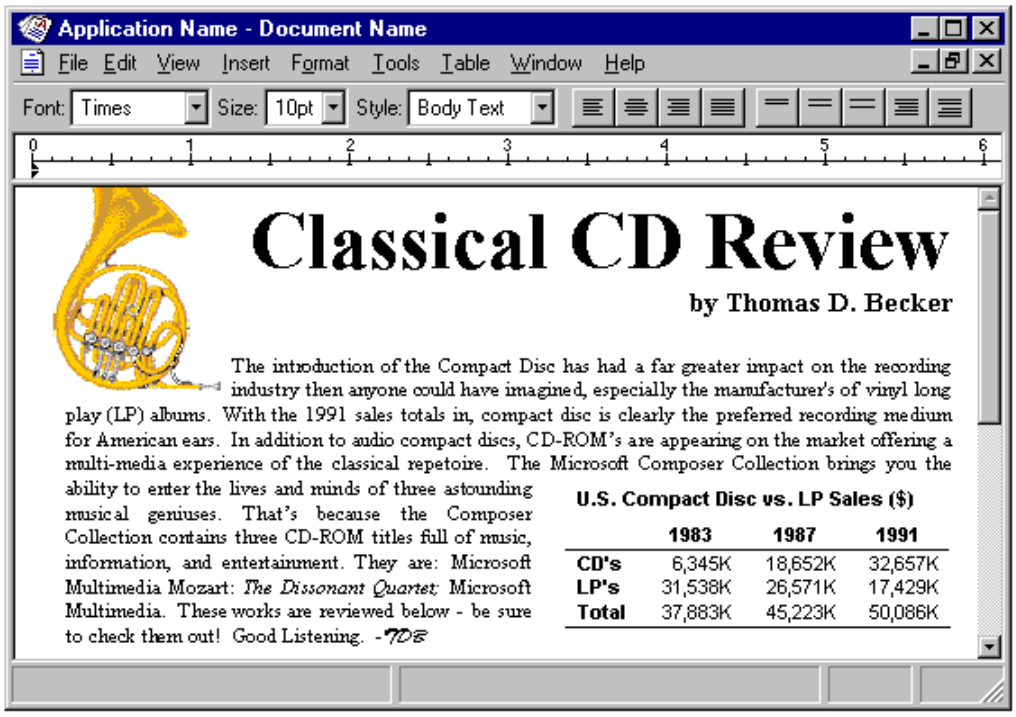


Figure 9.3 A maximized MDI child window

If the user maximizes one child window and then switches to another, display that window as maximized. Similarly, when the user restores one child window from its maximized state, restore all other child windows to their previous sizes.

Switching Between MDI Child Windows

Apply the same common mouse conventions for activating and switching between primary windows for MDI child windows. CTRL+F6 and CTRL+TAB (and SHIFT+ modified combinations to cycle backwards) are the recommended keyboard shortcuts for switching between child windows. In addition, include a Window menu on the menu bar of the parent window with commands for switching between child windows and managing or arranging the windows within the MDI parent window—for example, Tile or Cascade.

When the user switches child windows, you can change the interface of the parent window—such as its menu bar, toolbar, or status bar—to appropriately reflect the commands that apply to that child window. However, provide as much consistency as possible, keeping constant any menus that represent the document files and control the application or overall parent window environment, such as the File menu or the Window menu.

MDI Alternatives

MDI does have its limitations. MDI reinforces the visibility of the application as the primary focus for the user. While the user can start an MDI application by directly opening one of its document or data files, to work with multiple documents within the same MDI parent window, the user must use the application's interface for opening those documents.

When the user opens multiple files within the same MDI parent window, the storage relationship between the child windows and the objects being viewed in those windows is not consistent. That is, while the parent window provides

visual containment for a set of child windows, it does not provide containment for the files those windows represent. This makes the relationship between the files and their windows more abstract.

Similarly, because the MDI parent window does not actually contain the objects opened within it, MDI cannot support an effective design for persistence. When the user closes the parent window and then reopens it, the context cannot be restored because the application state must be maintained independently from that of the files last opened in it.

MDI can make some aspects of the OLE interface unintentionally more complex. For example, if the user opens a text document in an MDI application and then opens a worksheet embedded in that text document, the task relationship and window management breaks down, because the embedded document's window does not appear in the same MDI parent window.

Finally, the MDI technique of managing windows by confining child windows to the parent window can be inconvenient or inappropriate for some tasks, such as designing with window or form layout tools. Similarly, the nested nature of child windows may make it difficult for the user to differentiate between a child window in a parent window versus a primary window that is a peer with the parent window, but positioned on top.

While MDI provides useful conventions for managing a set of related windows, it is not the only means of supporting task management. Some of its window management techniques can be applied in some alternative designs. The following—workspaces, workbooks, and projects—are examples of some of these design alternatives. They present a single window design model, but in such a way that preserves some of the window and task management benefits found in MDI.

While these examples suggest a form of containment, you can also apply some of these designs to display multiple views of the same data. Similarly, these alternatives provide greater flexibility with respect to the types of objects that they may contain. As with any container, you can define your implementation to hold and manage only certain types of objects. For example, an appointment book and an index card file are both containers that organize a set of information but may differ in the way and type of information they manage. Whether you define a container to hold the same or different types of objects depends on the design and purpose of the container.

> **Note** The following examples illustrate alternatives of data-centered window or task management. They are not exclusive of other possible designs. They are intended only as suggestive possibilities, rather than standard constructs. As a result, the system does not include these constructs and provides no explicit programming interfaces. In addition, some specific details are left to you to define.

Workspaces

A *workspace* shares many of the characteristics of MDI, including the association and management of a set of related windows within a parent window, and the sharing of the parent window's interface elements, such as menus, toolbars, and status bar. Figure 9.4 shows an example of a workspace.

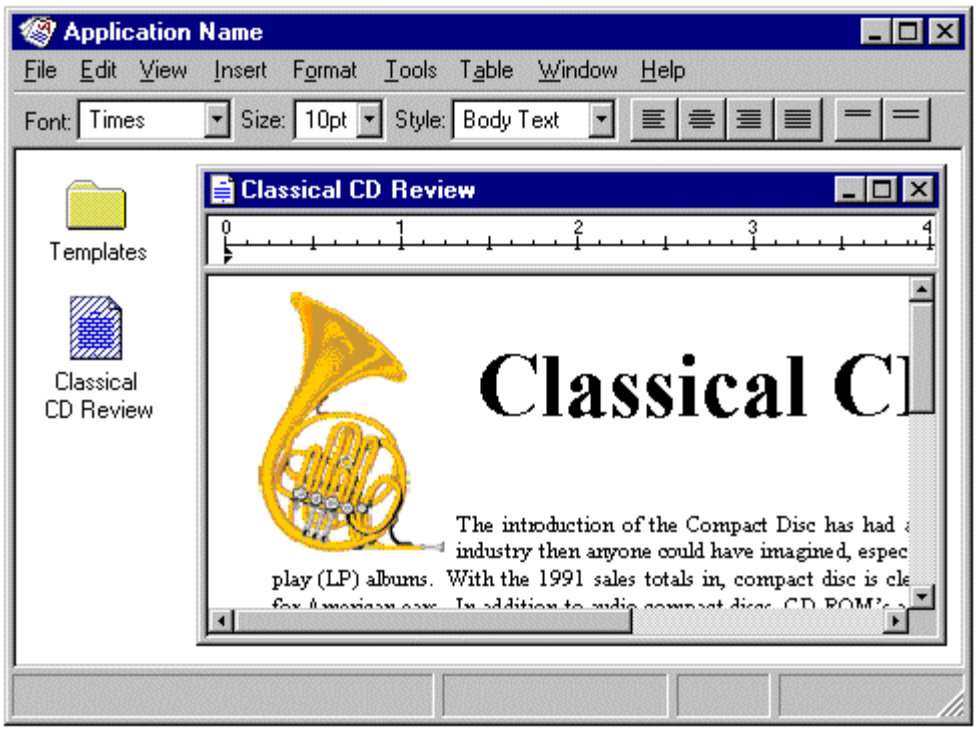


Figure 9.4 A workspace

Workspaces as a Container

Based on the metaphor of a work area, like a table, desktop, or office, a workspace differs from an MDI by including the concept of containment. Objects, represented as icons, may be contained or stored in the workspace in the same way they can be contained in folders. However, within a workspace, you open icons as child windows within the workspace parent window. In this way, a workspace's behavior is similar to that of the desktop, except that a workspace can be displayed as an icon and opened into a window. To have an object's window appear in the workspace, its icon must reside there. To facilitate this, you provide interaction techniques, such as drag and drop, that allow the user to move, copy, or link icons into the workspace.

The workspace is an object itself and therefore you define its specific commands and properties. Using information based on the registry, you also include commands for creating new objects within the workspace and, optionally, a Save All command that saves the state of all the objects opened in the workspace.

Workspaces for Task Grouping

Because a workspace visually contains and constrains the icons and windows of the objects placed in it, you can define workspaces to allow the user to organize a set of objects for particular tasks. Like an MDI, this makes it easy for the user to move or switch to a set of related windows as a set.

Also similar to an MDI, the child windows of objects opened in the workspace can share the interface of the parent window. For example, if the workspace includes a menu bar, the windows of any objects contained within the workspace share the menu bar. If the workspace does not have a menu bar, or if you provide an option for the user to

hide the menu bar, the menu bar appears within the document's child window. The parent window can also provide a framework for sharing toolbars and status bars.

A workspace provides support for the user to move the icons from the workspace into other containers, such as the desktop and folders. When the user moves an icon out of the workspace, places it on the desktop or in a folder, and then opens it, it appears in its own window and does not open a workspace window. Its interface elements, such as its menu bar, also appear within its own window. Only when the icon is stored in a workspace does it share the workspace menu bar.

Window Management in a Workspace

A workspace manages windows using the same conventions as MDI. When a workspace closes, all the windows within it close. You retain the state of these windows, for example, their size and position within the workspace, so you can restore them when the user reopens the workspace.

Like most primary windows, when the user minimizes the workspace window, the window disappears from the screen but its entry remains in the taskbar. Minimized windows of icons opened within the workspace have the same behavior and appearance as minimized MDI child windows. Similarly, maximizing a window within a workspace can follow the MDI technique: if the window's maximize size exceeds the size of the workspace window, the child window merges with the workspace window and its title bar icon and window buttons appear in the menu bar of the workspace window.

A workspace always provides a means of navigating between the child windows within a workspace, such as listing the open child windows on a Window drop-down menu and on the pop-up menu for the parent window, in addition to simple window activation.

Workbooks

A *workbook*, as shown in Figure 9.5, is another alternative for managing a set of views — one which uses the metaphor of a book or notebook instead of a work area. Within the workbook, you present views of objects as sections within the workbook's primary window rather than in individual child windows.

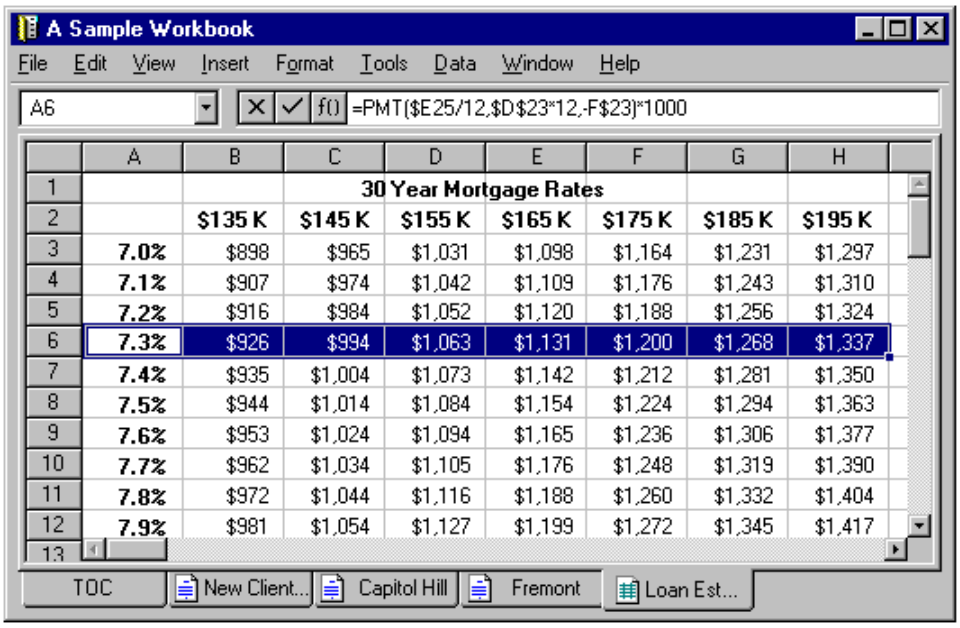


Figure 9.5 A workbook

For a workbook, tabs serve as a navigational interface to move between different sections. Each section represents a view of data, such as an individual document. Unlike a folder or workspace, you use a workbook for ordered content; that is, where the order of the sections has significance. In addition, you can optionally include a special section listing the content of the workbook, like a table of contents. This view can also be used as part of the navigational interface for the workbook.

A workbook shares an interface similar to an MDI parent window with all of its child windows maximized. The sections can share the parent window's interface elements, such as the menu bar and status bar. When the user switches sections within the workbook, the menu bar changes so that it applies to the current object. When the user closes a workbook, follow the common conventions for handling unsaved edits or unapplied transactions when any primary window closes.

For more information about the conventions for closing a primary window, see Chapter 6, "Windows."

Support transfer operations so that the user can move, copy, and link objects into the workbook. Also provide an Insert command that allows the user to create new objects, including a new tabbed section in the workbook. You can also include a Save All command, which saves any uncommitted changes or prompts the user to save or discard those changes.

Projects

A *project*, shown in Figure 9.6, is another window management technique that provides for association of a set of objects and their windows, but without visually containing the windows. A project is similar to a folder in that the icons contained within it can be opened into windows that are peers with the parent window. As a result, each child window also has its own entry in the taskbar. Furthermore, a project provides window management for the windows of its content, unlike a folder. For example, when the user opens a document in a folder and then closes the folder, it

has no effect on the window of the opened document. However, when the user closes a project window, all the child windows of objects contained in the project also close. In addition, when the user opens a project window, this action restores the windows of objects contained within it to their previous state.

Similarly, when the user minimizes a project window, this action minimizes any windows of the objects it contains. Taskbar entries for these windows remain. Allow the user to restore a specific child window without restoring the project window or other windows within the project. In addition, support the user independently minimizing any child window without affecting the project window.

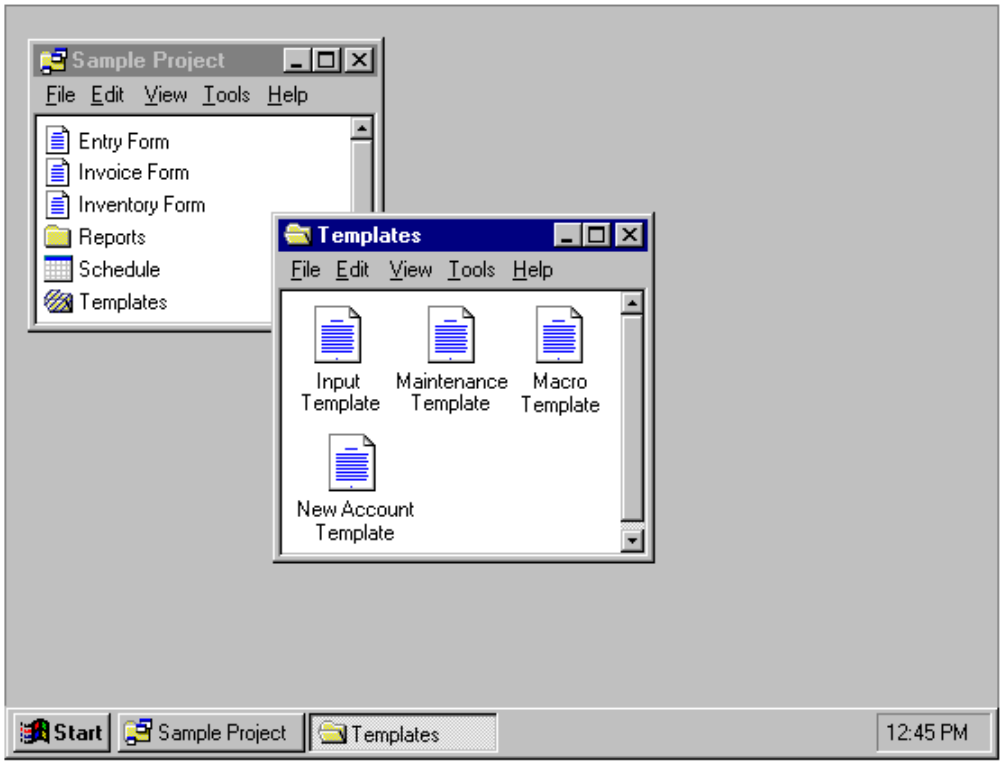


Figure 9.6 A project

Do not allow the windows of objects stored in the project to share the menu bar or other areas within the project window. Instead, include the interface elements for each object in its own window. As another option, you can provide toolbar palette windows that can be shared among the windows of the objects in the project.

Just as in workspaces and workbooks, a project should include commands for creating new objects within the project, for transferring objects in and out of the project, and for saving any changes for the objects stored in the project. In addition, a project should include commands and properties for the project object itself.

Selecting a Window Model

Deciding how to present your application's collection of related tasks or processes requires considering a number of design factors: presentation of object or task, screen layout, data-centered design, and the combining of alternatives.

Presentation of Object or Task

What an object represents and how it is used and relates to other objects influences how you present its view. Simple objects that are self contained may not require a primary window, or only require a set of menu commands and a property sheet to edit their properties.

An object with user-accessible content in addition to properties, such as a document, requires a primary window. This single document window interface can be sufficient when the object's primary presentation or use is as a single unit, even when containing different types. Alternative views can easily be supported with controls that allow the user to change the view. Simple simultaneous views of the same data can even be supported by splitting the window into panes. The system uses the single document window style of interface for most of the components it includes, such as folders.

MDI, workspaces, workbooks, and projects work better when the composition of an object requires simultaneous views or the nature of the user's tasks requires views of multiple objects. These constructs provide a focus for specific user activities, within the larger environment of the desktop.

You can use an MDI for viewing homogeneous types. The user cannot mix different objects within the same MDI parent windows unless you supply them as part of the application. On the other hand, MDI can provide compound documents that support views of different types of objects.

Use a workbook when you want to optimize quick user navigation of multiple views. A workbook simplifies the task by eliminating the management of child windows, but in doing so, it limits the user's ability to see simultaneous views.

Workspaces and projects provide flexibility for viewing and mixing of objects and their windows. Use a workspace as you would an MDI, when you want to clearly segregate the icons and their windows used in a task. Use a project when you do not want to constrain any child windows.

A project provides the greatest flexibility for user placement and arrangement of its windows. It does so, however, at the expense of an increase in complexity because it may be more difficult for a user to differentiate the child window of a project from windows of other applications.

Display Layout

Consider the requirements for layout of information. For very high resolution displays, the use of menu bars, toolbars, and status bars poses little problem for providing adequate display of the information being viewed in a window. Similarly, the appearance of these common interface elements in each window has little impact on the overall presentation. At VGA resolution, however, this can be an issue. The interface components for a set of windows should not affect the user's work area so that the user cannot easily view or manipulate their data.

MDI, workspaces, workbooks, and projects all allow interface components to be shared among multiple views. Within shared elements, it must be clear when a particular interface component applies. While you can automatically switch the content of those components, consider what functions are common across views or child windows and present them in a consistent way to provide for stability in the interface. For example, if multiple views share a Print toolbar button, present that button in a consistent location. If the button's placement constantly shifts when the user switches the view, the user's efficiency in performing the task decreases. In addition, shared interfaces may make user customization of interface components more complex because you need to indicate whether the customization applies to the current context or across all views.

Regardless of the window model you chose, always consider allowing users to determine which interface components they wish to have displayed. Doing so means that you also need to consider how to make basic functionality available if the user hides a particular component. For example, pop-up menus can often be used to supplement the interface when the user hides the menu bar.

Data-Centered Design

A single document window interface provides the best support for a simple, data-centered design; MDI supports a more conventional application-centered design. It is best suited to multiple views of the same data or contexts where the application does not represent views of user data. You can use workspaces, workbooks, and projects to provide single document window interfaces while preserving some of the management techniques provided by MDI.

Combination of Alternatives

Single document window interfaces, MDIs, workspaces, workbooks, and projects are not necessarily exclusive design techniques. It may be advantageous to combine these techniques. For example, documents can be presented within a workspace. You can also design workbooks and projects as objects within a workspace. In similar fashion, a project might contain a workbook as one of its objects.

CHAPTER 10

Integrating with the System

In addition to providing standard interface components such as windows and controls, Microsoft Windows includes special support that allows you to design your application to fit and operate effectively with the system. Users appreciate seamless integration between the system and their applications. This chapter covers details about integrating your software with the system and how to extend its features. Some of the conventions and features may not be supported in all releases. For more information about specific releases, see Appendix D, "Supporting Windows 95 and Windows NT Version 3.51."

The Registry

Windows provides a special repository called the *registry* that serves as a central configuration database for user-, application-, and computer-specific information. Although the registry is not intended for direct user access, the information placed in it affects your application's user interface. Registered information determines the icons, commands, and other features displayed for files. The registry also makes it easier to manage and support configuration information used by applications and eliminates redundant information stored in different locations.

The registry is a hierarchical structure. Each node in the tree is called a key. Each key can contain a subkey and data entries called values. Key names cannot include a space, backslash (\), or wildcard character (* or ?). Key names beginning with a period (**.**) are reserved for special syntax (for example, filename extensions), but you can include a period within the key name. The name of a subkey must be unique with respect to its parent key. Key names are not localized into other languages, although their values may be.

A key can have any number of values. A value entry has three parts: the name of the value, its data type, and the value itself. A value entry cannot be larger than 1 megabyte.

Note The example registry entries in this chapter represent only the hierarchical relationship of the keys. For more information about the registry and registry file formats, see the documentation in the Microsoft Win32 Software Development Kit.

When the user installs your application, register keys for where application data is stored, for filename extensions, icons, shell commands, OLE registration data, and for any special extensions. To register your application's information, you can create a registration (.REG) file and use the Registry Editor to merge this file into the system registry. You may also use other utilities that support this function, or use the system-supplied registry functions to access or manipulate registry data.

Note To use memory most efficiently, the system stores only the registry entries that have been installed and that are required for operation. Applications should never fail to write a registry entry because it is not already installed. To ensure this happens, use registry creation functions when adding an entry.

Registering Application State Information

Use the registry to store state information for your application. Typically, the data you store here will be information you may have stored in .INI files in previous releases of Windows. Create subkeys in the **HKEY_LOCAL_MACHINE** and **HKEY_CURRENT_USER** keys that include information about your application.

```
HKEY_LOCAL_MACHINE
                SOFTWARE
                    CompanyName
                        ProductName
                            Version
...
HKEY_CURRENT_USER
                SOFTWARE
                    CompanyName
                        ProductName
                            Version
```

Use your application's **HKEY_LOCAL_MACHINE** entry as the location to store computer-specific data and the **HKEY_CURRENT_USER** entry to store user-specific data. You can define your own structure for the information you store under your application's subkey.

Save your application's state whenever the user closes its primary window. In most cases, it is best to restore a window to its previous state when the user reopens it.

When the user shuts down the system with your application's window open, you may optionally store information in the registry so that the application's state is restored when the user starts up Windows. (The system does this for folders.) To have your application's state restored, store your window and application state information under its registry entries when the system notifies your application that it is shutting down. Store the state information in **HKEY_CURRENT_USER** and add a value name–value pair to the **RunOnce** subkey that corresponds to the command line string that will run your application. Then, allow it to restore its state when the same user next starts the system.

```
HKEY_LOCAL_MACHINE
               Software
                   Microsoft
                       Windows
                           CurrentVersion
                               RunOnce application/file identifier = command line
```

If you have multiple instances open, you can include value name entries for each or consolidate them as a single entry and use command-line switches that are most appropriate for your application. For example, you may include entries like the following.

```
WordPad Document 1 = C:\Program Files\Wordpad.exe Letter to Bill /restore
WordPad Document 2 = C:\Program Files\Wordpad.exe Letter to Paul /restore
Paint = C:\Program Files\Paint.exe Abstract.bmp Cubist.bmp
```

As long as you provide a valid command-line string that your application can process, you can format the entry in a way that best fits your application.

Note The system's ability to restore an application's state depends on whether the application and its data files are still available. If they have been deleted or the user has logged in over the network, where the same files are not available, the system cannot restore the state.

You can also include a **RunOnce** entry under the **HKEY_LOCAL_MACHINE** key. When using this entry, however, the system runs the application before starting up. You can use this entry for applications that may need to query the user for information that affects how Windows starts. Just remember that any entry here will affect all users of the computer.

RunOnce entries are automatically removed from the registry once the system starts up. Therefore, you need not remove or update the entries, but your application must always save its state when the user shuts down the system. The system also supports a **Run** subkey in both the **HKEY_CURRENT_USER** and **HKEY_LOCAL_MACHINE** keys. The system runs any value name entries under this subkey after the system starts up, but does not remove those entries from the registry. For example, a virus check program can be installed to run automatically after the system starts up.

Registering Application Path Information

The system supports "per application" paths. If you register a path, Windows sets the PATH environment to be the registered path when it starts your application. You set your application's path in the **App Paths** subkey under the **HKEY_LOCAL_MACHINE** key. Create a new key using your application's executable (.EXE) filename as its name. Under this key, create a value name called Default, setting its value to the path of your executable file. The system uses this entry to locate your application, if it fails to find it in the current path; for example, if the user chooses the Run command on the Start menu and only includes the filename of the application, or if a shortcut icon doesn't include a path setting. If you need to place dynamic-link libraries (DLLs) in a separate directory, you can also include another value entry called Path and set its value to the path of your DLL files.

```
HKEY_LOCAL_MACHINE
               Software
                   Microsoft
                       Windows
                           CurrentVersion
                               App Paths
                                   Application Executable Filename
                                       Default = path
                                       Path = path
```

The system will automatically update the path and default entries if the user moves or renames the application's executable file using the standard system user interface.

Register any systemwide shared dynamic-link libraries in a subkey under a **SharedDLLs** subkey of **HKEY_LOCAL_MACHINE** key. If the file already exists, increment the entry's usage count index. For more information about the usage count index, see the "Installation" section in this chapter.

```
HKEY_LOCAL_MACHINE
               Software
                   Microsoft
                       Windows
                           CurrentVersion
                               SharedDLLs filename [= usage count index]
```

Registering File Extensions

If your application creates and maintains files, register entries for the file types that you expose directly to users and that you want users to be able to easily differentiate. For every file type you register, include at least two entries: a filename-extension key entry and a class-definition key entry. If you do not register an extension for a file type, it will be displayed with the system's generic file object icon, as shown in Figure 10.1, and its extension will always be displayed. In addition, the user will not be able to double-click the file to open it. (Open With will be the icon's default command.)

Figure 10.1 System-generated icons for unregistered types

The Filename Extension Key

The filename extension entry maps a filename extension to an application identifier. To register an extension, create a subkey in the **HKEY_CLASSES_ROOT** key using the three-letter extension (including a period) and set its value to an application identifier.

```
HKEY_CLASSES_ROOT
        .ext = ApplicationIdentifier
```

For the value of the application identifier (also known as programmatic identifier or Prog ID), use a string that uniquely identifies a given class. This string is used internally by the system and is not exposed directly to users (unless explicitly exported with a special registry utility); therefore, you need not localize this entry.

Multiple extensions may have the same application identifier. To ensure that each file type can be distinguished by the user, however, define each extension such that each has a unique application identifier, unless you application already supports multiple extensions or you have utility files that the user does not interact with directly. The system provides no arbitration for applications that use the same extensions. So define unique identifiers and check the registry to avoid writing over and replacing existing extension entries, a practice which may seriously affect the user's existing files. More specifically, avoid registering an extension that conflicts or redefines the common filename extensions used by the system; these extensions are displayed in Table 10.1.

Table 10.1 Common Filename Extensions Supported by Windows

Extension	Type description
386	Windows virtual device driver
3GR	Screen grabber for MS-DOS–based applications
ACM	Audio compression manager driver
ADF	Administration configuration files
ANI	Animated pointer
AVI	Video clip
AWD	Fax viewer document
AWP	Fax key viewer
AWS	Fax signature viewer
BAK	Backed-up file
BAT	MS-DOS batch file
BFC	Briefcase
BIN	Binary data file
BMP	Picture (Windows bitmap)
CAB	Windows Setup file
CAL	Windows Calendar file
CDA	CD audio track
CFG	Configuration file
CNT	Configuration file
CNT	Help contents
COM	MS-DOS application
CPD	Fax cover page
CPE	Fax cover page
CPI	International code page
CPL	Control Panel object
CRD	Windows Cardfile document
CSV	Command-separated data file
CUR	Cursor (pointer)

Table 10.1 Common Filename Extensions Supported by Windows (*continued*)

Extension	Type description
DAT	System data file
DCX	Fax viewer document
DLL	Application extension (Dynamic-link library)
DOC	WordPad document
DOS	MS-DOS file (also extension for NDIS2 net card and protocol drivers)
DRV	Device driver
EXE	Application
FND	Saved search
FON	Font file
FOT	Shortcut to font
GR3	Windows 3.0 screen grabber
GRP	Program group file
HLP	Help file
HT	HyperTerminal file
ICM	ICM profile
ICO	Icon
IDF	MIDI instrument definition
INF	Setup information
INI	Configuration settings
KBD	Keyboard layout
LGO	Windows logo driver
LIB	Static-link library
LNK	Shortcut
LOG	Log file
MCI	MCI command set
MDB	File viewer extension
MID	MIDI sequence
MIF	MIDI instrument file
MMF	Microsoft Mail message file

Table 10.1 Common Filename Extensions Supported by Windows (*continued*)

Extension	Type description
MMM	Animation
MPD	Mini-port driver
MSG	Microsoft Exchange mail document
MSN	The Microsoft Network home base
MSP	Windows Paintbrush picture
NLS	Natural language services driver
PAB	Microsoft Exchange personal address book
PCX	Picture (PCX format)
PDR	Port driver
PF	ICM profile
PIF	Shortcut to MS-DOS–based application
PPD	PostScript® printer description file
PRT	Printer formatted file (result of Print to File option)
PST	Microsoft Exchange personal information store
PWL	Password list
QIC	Backup set for Microsoft Backup
REC	Windows Recorder file
REG	Application registration file
RLE	Picture (RLE format)
RMI	MIDI sequence
RTF	Document (rich text format)
SCR	Screen saver
SET	File set for Microsoft Backup
SHB	Shortcut into a document
SHS	Scrap
SPD	PostScript printer description file
SWP	Virtual memory storage
SYS	System file

Table 10.1 Common Filename Extensions Supported by Windows (*continued*)

Extension	Type description
TIF	Picture (TIFF format)
TMP	Temporary file
TRN	Translation file
TSP	Windows telephony service provider
TTF	TrueType® font
TXT	Text document
VBX	Visual Basic® control file
VER	Version description file
VXD	Virtual device driver
WAV	Sound wave
WPC	WordPad file converter
WRI	Windows Write document
XAB	Microsoft Mail address book

Also investigate extensions commonly used by popular applications so you can avoid creating a new extension that might conflict with them, unless you intend to replace or superset the functionality of those applications

The Application Identifier Key

The second registry entry you create for a file type is its class-definition (Prog ID) key. Using the same string as the application identifier you used for the extension's value, create a key and assign a type name as the value of the key.

```
HKEY_CLASSES_ROOT
              .ext = ApplicationIdentifier
       ApplicationIdentifier = Type Name
```

Under this key, you specify shell and OLE properties of the class. Provide this entry even if you do not have any extra information to place under this key; doing so provides a better label for users to identify the file type. In addition, you use this entry to register the icon for file type.

Define the type name (also known as the MainUserTypeName) as the human-readable form of its application identifier or class name. It should convey to the user the object's name, behavior, or capability. A type name can include all of the following elements:

1. *Company Name*
   Communicates product identity.

2. *Application Name*
   Indicates which application is responsible for activating a data object.

3. *Data Type*
   Indicates the basic category of the object (for example, drawing, spreadsheet, or sound). Limit the number of characters to a maximum of 15.

4. *Version*
   When there are multiple versions of the same basic type, for upgrading purposes, a version number is necessary to distinguish types.

When defining your type name, use title capitalization. The name can include up to a maximum of 40 characters. Use one of the following three recommended forms:

1. *Company Name Application Name*[*Version*] *Data Type*
   For example, Microsoft Excel 5.0 Worksheet.

2. *Company Name-Application Name*[*Version*] *Data Type*
   For cases when the company name and application are the same—for example, ExampleWare 2.0 Document.

3. *Company Name Application Name* [*Version*]
   When the application sufficiently describes the data type—for example, Microsoft Graph 3.0.

These type names provide the user with a precise language for referring to objects. Because object type names appear throughout the interface, the user becomes conscious of an object's type and its associated behavior. However, because of their length, you may also want to include a short type name. A *short type name* is the data type portion of the full type name. Applications that support OLE always include a short type name entry in the registry. Use the short type name in drop-down and pop-up menus. For example, a Microsoft Excel 5.0 Worksheet is simply referred to as a "Worksheet" in menus.

To provide a short type name, add an **AuxUserType** subkey under the application's registered **CLSID** subkey (which in turn is under the **CLSID** key).

```
HKEY_CLASSES_ROOT
                .ext = ApplicationIdentifier
...
        ApplicationIdentifier = Type Name
                    CLSID = {CLSID identifer}
...
                CLSID
                    {CLSID identifer}
                        AuxUserType
                            2 =Short Type Name
```

For more information about registering type names, see the *Microsoft OLE Programmer's Reference*.

If a short type name is not available for an object because the string was not registered, use the full type name instead. All controls that display the full type name must allocate enough space for 40 characters in width. By comparison, controls need only accommodate 15 characters when using the short type name.

Supporting Creation

The system supports the creation of new objects in system containers such as folders and the desktop. Register information for each file type that you want the system to include. The registered type will appear on the New command for a folder and the desktop. This provides a more data-centered design because the user can create a new object without having locate and run the associated application.

To register a file type for inclusion, create a **ShellNew** subkey under the extension's subkey in **HKEY_CLASSES_ROOT**.

```
HKEY_CLASSES_ROOT
                .ext = ApplicationIdentifier
                    ShellNew = Value
```

Assign a value entry to the **ShellNew** subkey with one of the four methods for creating a file with this extension:

Value name	Value	Result
NullFile	""	Creates a new file of this type as a null (empty) file.
Data	*binary data*	Creates a new file containing the binary data.
FileName	*path*	Creates a new file by copying the specified file.
Command	*path*	Executes the command. Use this to run your own application code to create a new file (for example, run a wizard).

When using the Command value, you must register the name of the command. The system supports the display of any secondary windows you might want to display as a part of the creation process. The system also provides you with automatic filename creation for the new file.

Registering Icons

The system uses the registry to determine which icon to display for a specific file. You register an icon for every data file type that your application supports and that you want the user to be able to distinguish easily. Create a **DefaultIcon** subkey entry under the application identifier subkey you created and define its value as the filename containing the icon. Typically, you use the application's executable (.EXE) filename and the index of the icon within the file. The index value corresponds to the icon within the file. A positive number represents the icon's position in the file. A negative number corresponds to the inverse of the resource ID number of the icon. The icon for your application should always be the first icon resource in your .EXE file. The system always uses the first icon resource to represent executable files. This means the index value for your data files will be a number greater than 0.

```
HKEY_CLASSES_ROOT
               ApplicationIdentifier = Type Name
                   DefaultIcon = path [,index]
```

Instead of registering the application's .EXE file, you can register the name of a .DLL file, an icon (.ICO) file, or bitmap (.BMP) file to supply your data file icons. If an icon does not exist or is not registered, the system supplies an icon derived from the icon of the file type's registered application. If no icon is available for the application, the system supplies a generic icon. These icons do not make your files uniquely identifiable, however, so design icons for both your application and its data file types. Include the following sizes: 16×16 pixel (16 color) , 32×32 pixel (16 color), and 48×48 pixel (256 color).

For more information about designing icons, see Chapter 13, "Visual Design."

Registering Commands

Many of the commands found on icons, including Send To, Cut, Copy, Paste, Create Shortcut, Delete, Rename, and Properties, are provided by their container—that is, their containing folder or the desktop. But you must provide support for the icon's primary commands, also known as verbs, such as Open, Edit, Play, and Print. You can also specify your own commands that apply to your file types, such as a What's This? command. You may also add commands for existing file types.

For more information about context-sensitive Help for file types, see Chapter 12, "User Assistance."

To add these commands, in the HKEY_CLASSES_ROOT key, you register a **Shell** subkey and a subkey for each verb, and a Command subkey for each command name.

```
HKEY_CLASSES_ROOT
                ApplicationIdentifier = Type Name
                    Shell [ = default Verb [,Verb2 [,..]]
                        Verb [ = Menu Name]
                            Command = path [parameters]
                            ddeexec = DDE command string
                                Application = DDE Application Name
                                Topic = DDE topic name
```

A Verb is a language-independent name of the command. Applications may use it to invoke a specific command programmatically. The system defines Open, Print, Find, and Explore as standard Verbs and automatically provides menu names and access key assignments for these, localized in each international version of Windows. When you supply Verbs other than these, provide strings localized for the specific version of Windows on which the application is installed.

To support user execution of a Verb, provide the path for the application or a dynamic data exchange (DDE) command string. You can include command-line switches. For paths, include a %1 parameter. This parameter is an operational placeholder for whatever file the user selects.

You may have different values for each Verb. You may assign one application to carry out the Open command and another to carry out the Print command, or use the same application for all Verbs..

The menu command corresponding to the Verbs for a file type are displayed to the user, either on a folder's File drop-down menu or pop-up menu for a file's icon. These appear at the top of the menu. You define the order of the menu commands by ordering the Verbs in the value of the **Shell** key. The first Verb becomes the default command in the menu.

By default, capitalization follows how you enter format the menu name value of the **Verb** subkey. Although the system automatically capitalizes the standard commands (Open, Print, Explore, and Find), you can use the value of the menu name to format the capitalization differently. Similarly, you use the menu command value to set the access key for the menu command following normal menu conventions, prefixing the character in the name with an ampersand (&). Otherwise, the system sets the first letter of the command as the access key for that command.

Enabling Printing

If your file types are printable, include a Print Verb value in the Application Identifier\ Shell subkey under **HKEY_CLASSES_ROOT**, following the conventions described in the previous section. This will display the Print command on the pop-up menu for the icon and when the user selects the icon on the File menu of the folder in which the icon resides. When the user chooses the Print command, the selected object prints on the default printer.

Also register a Print To registry entry for the file types your application supports. This entry enables drag and drop of a file onto a printer icon. Although a Print To command is not displayed on any menu, the printer includes Print Here as the default command on the pop-up menu displayed when the user drag and drops a file on the printer using button 2.

In both cases, print the file, preferably, without opening the application's primary window. One way to do this is to provide a command-line switch that runs the application for handling the printing operation only (for example, `Word.exe /p`). Display some form of user feedback that indicates whether a printing process has been initiated and,

if so, its progress. For example, this feedback could be a modeless message box that displays, "Printing page *m* of *n* on *printer name*" and a Cancel button. You may also include a progress indicator control.

Registering OLE

Applications that support OLE use the registry as the primary means of defining class types, operations, and properties for data types supported by applications. You store OLE registration information in the **HKEY_CLASSES_ROOT** key in subkeys under the **CLSID** subkey and in the class description's (Prog ID) subkey.

For more information about the specific registration entries for OLE, see the *Microsoft OLE Programmer's Reference*.

Registering Shell Extensions

Your application can extend the functionality of the operational environment provided by the system, also known as the shell, in a number of ways. A *shell extension* enhances the system by providing additional ways to manipulate file objects, by simplifying the task of browsing through the file system, or by giving the user easier access to tools that manipulate objects in the file system.

Every shell extension requires a *handler*, a special application code (32-bit OLE in-proc server) that implements subordinate functions. The types of handlers you can provide include:

- Pop-up menu handlers: These add menu items to the context menu for a particular file type.
- Drag handlers: These allow you to support the OLE data transfer conventions for drag and drop operations of a specific file.
- Drop handlers: These allow you to execute some action when the user drops objects on a specific type of file.
- Nondefault drag and drop handlers: These are pop-up menu handlers that the system calls when the user drags and drops an object by using mouse button 2.
- Icon handlers: These can be used to add per-instance icons for file objects or to supply icons for all files of a specific type.
- Property sheet handlers: These add pages to a property sheet that the shell displays for a file object. The pages can be specific to a class of files or to a particular file object.
- Copy-hook handlers: These are called when a folder or printer object is about to be moved, copied, deleted, or renamed by the user. The handler can be used to allow or prevent the operation.

You register the handler for a shell extension in the **HKEY_CLASSES_ROOT\CLSID** key. Each CLSID contains a list of class identifier key values such as {00030000-0000-0000-C000-000000000046}. Each class identifier must also be a globally unique identifier.

For more information about handlers and class identifiers, see the *Microsoft OLE Programmer's Reference*.

Note Support for shell extensions may depend on the version of Windows installed. For more information about specific releases, see Appendix D, "Supporting Windows 95 and Windows NT 3.51."

You must also create a **shellex** subkey under the class description entry in the **HKEY_CLASSES_ROOT** key.

```
HKEY_CLASSES_ROOT
                ApplicationIdentifier = Type Name
                    Shell [ = default Verb [,Verb2 [,..]]
                        Verb [ = Menu Name]
                            Command = path [parameters]
                            ddeexec = DDE command string
                                Application = DDE Application Name
                                Topic = DDE topic name
                        shellex
                            HandlerType
                                {CLSID identifer} = Handler Name
                            ...
                            HandlerType = {CLSID identifier}
```

The shell also uses several other special keys, such as *****, **Folder**, **Drives**, and **Printers**, under **HKEY_CLASSES_ROOT**. You can use these keys to register extensions for system-supplied objects. For example, you may use the ***** key to register handlers that the shell calls whenever it creates a pop-up menu or property sheet for a file object, as in the following example.

```
HKEY_CLASSES_ROOT
               * = *
                  shellex
                     ContextMenuHandlers
                        {00000000-1111-2222-3333-0000000001}
                     PropertySheetHandlers = SummaryInfo
                        {00000000-1111-2222-3333-0000000002}
                     IconHandler = {00000000-1111-2222-3333-000000003}
```

The shell would use these handlers to add to the pop-up menus and property sheets of every file object. (The entries are intended only as examples, not literal entries.)

A pop-up menu handler may add commands to the pop-up menu of a file type, but it may not delete or modify existing menu commands. You can register multiple pop-up menu handlers for a file type. The order of the subkey entries determines the order of the items in the context menu. Handler-supplied menu items always follow registered command names.

When registering an icon handler for providing per-instance icons for a file type, set the value for the **DefaultIcon** key to %1. This denotes that each file instance of this type can have a different icon.

Supporting the Quick View Command

The system includes support for fast, read-only views of many file types when the user chooses the Quick View command from the file object's menu. This allows the user to view files without opening the application.

If your file type is not supported, install a file parser that translates your file type into a format the system file viewer can read. Place the file parser DLL into the Windows System directory and register your extension, in the **QuickView** subkey of the **HKEY_CLASSES_ROOT** key.

```
HKEY_CLASSES_ROOT
                QuickView
                    .EXT = File Type Name
```

Although this approach allows you to easily support viewers for your data file types, it limits the interaction options for your file types to those provided by the system. Alternatively, you can create your own file viewer, using the system-supplied interfaces. In this case, you need to register not only your extension in a **QuickView** subkey, but also include a CLSID identifier subkey under the extension key as well as in the **CLSID** key. You also include a key entry for defining the path for the file viewer DLL.

```
HKEY_CLASSES_ROOT
                QuickView
                    .EXT = File Type Name
                        {CLSID identifier} = File Viewer Name
                ...
                CLSID
                    {CLSID identifier} = File Viewer Name
                        InprocServer32 = path
```

To add your own file viewer for a file type already registered, follow the same procedure as for creating a file viewer for a new extension, except add your CLSID identifier to the existing entries under the extension subkey and under the **CLSID** subkey. Then, whenever an extension is registered, the system automatically adds Quick View to the command for the file object.

You can also support the command for objects stored within your application's interface, either by supplying a specific viewer for your data types or by writing the data to a temporary file and then executing a file viewer and passing the temporary file as a parameter.

Registering Sound Events

Your application can register specific events to which the user can assign sound files so that when those events are triggered, the assigned sound file is played. To register a sound event, create a key under the **HKEY_CURRENT_USER** key.

```
HKEY_CURRENT_USER
        AppEvents
                Event Labels
                        EventName = Event Name
```

Set the value for EventName to a human-readable name.

Installation

The following sections provide guidelines for installing your application's files. Applying these guidelines will help you reduce the clutter of irrelevant files when the user browses for a file. In addition, you'll reduce the redundancy of common files and make it easier for the user to update applications or the system software.

Copying Files

When the user installs your software, avoid copying any files into the Windows directory (folder) or its System subdirectory. Doing so clutters the directory and may degrade system performance. Instead, create a single directory, preferably using the application's name, in the Program Files directory (or the location that the user chooses). In this directory, place the executable file (.EXE). For example, if a program is named My Application, create a My Application directory and place My Application.exe in that directory.

Note To locate the Program Files directory, check the ProgramFilesDir value in the **CurrentVersion** subkey under the **HKEY_LOCAL_MACHINE** key. The actual directory may not literally be named Program Files. For example, in international versions of Windows the directory name is appropriately localized. For networks that do not support the Windows long filename conventions, MS-DOS names may be used instead.

In your application directory, create a subdirectory named System and place in it all support files that the user does not directly access, such as dynamic-link libraries and Help files (.HLP). For example, place a support file called My Application.dll in Program Files\My Application\System. Hide the support files and your application's System directory and register its location using a Path value in the **App Paths** subkey under the **HKEY_LOCAL_MACHINE** key. Although you may place support files in the same directory as your application, placing them in a subdirectory helps avoid confusing the user and makes files easier to manage.

Applications can share common support files to reduce the amount of disk space consumed by duplication. If some non–user-accessed files of your application are shared as systemwide components (such as Microsoft Visual Basic's Vbrun300.dll), place them in the System subdirectory of the directory where the user installs Windows. The process for installing shared files includes these logical steps:

1. Before copying the file, verify that it is not already present.
2. If the file is already present, compare its date and size to determine whether it is the same version as the one you are installing. If it is, increment the usage count in its corresponding registry entry.
3. If the file you are installing is not newer, do not overwrite the existing version.
4. If the file is not present, copy it to the directory.

If you store a new file in the Windows\System directory, register a corresponding entry in the **SharedDLL** subkey under the **HKEY_LOCAL_MACHINE** key.

If a file is shared, but only among your applications, create a subdirectory using your application's name in the Common Files subdirectory of the Program Files subdirectory and place the file there. Alternatively, for "suite"-style when multiple applications are bundled together, you can create a suite subdirectory in Program Files, where the executable files reside, and a System subdirectory with the support files shared only within the suite. In either case, register the path using the **Path** subkey under the **App Paths** subkey

Note To locate the Common Files directory, check the CommonFilesDir value in the **CurrentVersion** subkey of **HKEY_LOCAL_MACHINE**.

When installing an updated version of the shared file, ensure that it is upwardly compatible before replacing the existing file. Alternatively, you can create a separate entry with a different filename (for example, Vbrun301.dll).

Name your executable file, DLL files, and any other files that the user does not directly use, but that may be shared on a network, using conventional MS-DOS (8.3) names rather than than long filenames. This will provide better support for users operating in environments where these files may need to be installed on network services that do not support the Windows long filename conventions.

Windows no longer requires Autoexec.bat and Config.sys files. Ensure that your application also does not require these files. Because the system now supports loading device drivers when the application starts, they no longer need to be loaded through Config.sys when starting the system. Similarly, because the registry allows you to register your application paths, your application does not require path information in Autoexec.bat.

In addition, do not make entries in Win.ini. Storing information in this file can make it difficult to update or move your application. Instead, use the registry. The registry provides conventions for defining the location for most

application and user settings. If you have additional information that you do not want to put in the registry, you may create your own .INI file in your application's System directory.

Make certain you register the types supported by your application and the icons for these types along with your application's icons. In addition, register other application information, such as information required to enable printing.

For more information about the registry, see the section,"The Registry," earlier in this chapter.

Making Your Application Accessible

To make your application easily accessible to users, place a shortcut icon to the application in the Start Menu\Programs folder located in the main Windows directory. This adds the entry to the submenu of the Programs menu of the Start button. Avoid adding entries for every application you might include in your software; this quickly overloads the menu. Optionally, you can allow the user to choose which icons to place in the menu. Avoid defining a shortcut to a folder as your entry in the Start menu, as this creates a multilevel hierarchy. Including a single entry makes it easier and simpler for a user to access your application.

Note Similarly, you can still create a "program group" folder in the Programs folder using the Windows 3.1 dynamic date exchange (DDE) application programming interface (API). However, this is no longer recommended for applications installed on Windows 95 and later releases.

Also consider the layout of files you provide with your application. Windows now provides much greater flexibility for you to organize your files than did the Windows 3.1 Program Manager. In addition to the recommended structure for your main executable file and its support files, you may want to create special folders for documents, templates, conversion tools, or other files that the user accesses directly .

Designing Your Installation Program

Your installation program should offer the user different installation options such as:

- Typical Setup: Installation that proceeds with the common defaults set, copying only the most common files. Make this the default setup option.
- Compact Setup: Installation of the minimum files necessary to operate your application. This option is best for situations where disk space must be conserved—for example, on laptop computers. You can optionally add a Portable setup option for additional functionality designed especially for configurations on laptops, portables, and portables used with docking stations.
- Custom Setup: Installation for the experienced user. This option allows the user to choose where to copy files and which options or features to include. This can include options or components not available for compact or typical setup.
- CD-ROM Setup: Installation from a CD-ROM. This option allows users to select what files to install from the CD and allows them to execute the remaining files directly from the CD.
- Silent Setup: Installation using a command-line switch. This allows your setup program to run with a batch file

In addition to these setup options, your installation program should be a well-designed, Windows-based application and follow the conventions detailed elsewhere in this guide and in the following guidelines:

- Supply a common response to every option so that the user can step through the installation process by confirming the default settings (that is, by pressing the ENTER key).
- Tell users how much disk space they will need before proceeding with installation. In the custom setup option, adjust the figure as the user chooses to include or exclude certain options. If there is not sufficient disk space, let the user know, but also give the user the option to override.
- Offer the user the option to quit the installation before it is finished. Keep a log of the files copied and the settings made so the canceled installation can be cleaned up easily.
- Ask the user to install a disk only once during the installation. Lay out your files on disk so that the user does not have to reinsert the same disk multiple times.
- Provide an audio cue when the user needs to insert the next disk.
- Support installation from any location. Do not assume that installation must be done from a logical MS-DOS drive (such as drive A). Design your installation program to support any valid UNC path..
- Provide a progress indicator message box to inform the user how far they are through the installation process.

If you are creating your own installation program, consider using the wizard control. Using this control and following the guidelines for wizards will result in a consistent interface for users.

For more information about designing wizards, see Chapter 12, "User Assistance."

Naming your installation program Setup.exe or Install.exe will allow the system to recognize the file. When the user chooses the Install button in the Add/Remove Programs utility in the Control Panel, the system will automatically load and carry out your installation program.

Uninstalling Your Application

The user may need to remove your application to recover disk space or to move the application to another location. To facilitate this, provide an uninstall program with your application that removes its files and settings. Your uninstall program should follow the conventions detailed elsewhere in this guide and in the following guidelines:

- Display a window that provides the user with information about the progress of the uninstall process. You can also provide an option to allow the program to uninstall "silently"—that is, without displaying any information so that it can be used in batch files.
- Display clear and helpful messages for any errors your uninstall program encounters during the uninstall process.
- When uninstalling an application, decrement the usage count in the registry for any shared component—for example, a DLL. If the result is zero, give the user the option to delete the shared component with the warning that other applications may use this file and will not work if it is missing.

Registering your uninstall program will display your application in the list of the Uninstall page of the Add/Remove Program utility included with Windows. To register your uninstall program, add the following entry for your application:

```
HKEY_LOCAL_MACHINE
            SOFTWARE
                Microsoft
                    Windows
                        CurrentVersion
                            Uninstall
                                ApplicationName
                                    DisplayName = Application Name
                                    UninstallString = path [ switches ]
```

Both the **DisplayName** and **UninstallString** keys must be supplied and be complete for your uninstall program to appear in the Add/Remove Program utility. The path you supply to **UninstallString** must be the complete command line used to carry out your uninstall program. The command line you supply should carry out the uninstall program directly rather than from a batch file or subprocess.

Note Microsoft Windows NT 3.5 does not support an Add/Remove Programs utility. For information about provided compatibility with this version of Windows, see Appendix D, "Supporting Windows 95 and Windows NT Version 3.51."

Installing Fonts

When installing fonts with your application on a local system, determine whether the font is already present. If it is, rename your font file—for example, by appending a number to the end of its filename. After copying a font file, register the font in the **Fonts** subkey.

Installing Your Application on a Network

If you create a client-server application so that multiple users can access it from a network server, create separate installation programs: an installation program that allows the network administrator to prepare the server component of the application, and a client installation program that installs the client component files and sets up the settings to connect to the server. Design your client software so that an administrator can deploy it over the network and have it automatically configure itself when the user starts it.

For more information about designing client-server applications, see Chapter 14, "Special Design."

Because Windows may itself be configured to be shared on a server, do not assume that your installation program can store information in the main Windows directory on the server. In addition, shared application files should not be stored in the "home" directory provided for the user.

Design your installation program to support UNC paths. Also, use UNC paths for any shortcut icons you install in the Start Menu folder.

Supporting Auto Play

Window supports the ability to automatically run a file when the user inserts removable media that support insertion notification, such as CD-ROM, PCMCIA hard disks, or flash ROM cards. To support this feature, include a file named Autorun.inf in the root directory of the removable media. The contents of the file include the name of the file to carry out. Use the following form.

```
[autorun]
open = filename
```

Unless you specify a path, the system looks for the file in the root of the inserted media. If you want to run a file located in a subdirectory, include a path relative to the root; include that path with the file. For example:

```
open = My Directory\My File.exe
```

Running the file from a subdirectory does not change the current directory setting. The command-line string you supply can also include parameters or switches.

Because the autoplay feature is intended to provide automatic operation, design the file you specify in the Autorun.inf file to provide visual feedback quickly to confirm the successful insertion of the media. Consider using a startup up window with a graphic or animated sequence. If the process you are automating requires a long load time or requires user input, offer the user the option to cancel the process.

Although you can use this feature to install an application, avoid writing files to the user's local disk without first getting the user's confirmation. Even when you get the user's confirmation, minimize the file storage requirements, particularly for CD-ROM games or educational applications. Consuming a large amount of local file space defeats some of the benefits of the turnkey operation that the autoplay feature provides. Also, because a network administrator or the user may disable this feature, avoid depending on it for any required operations.

You can define the icon that the system displays for the media by including an entry in in the Autorun.inf file that includes the filename (and optionally path) file including the icon using the following form.

```
icon = filename
```

The filename can specify an icon (.ICO), a bitmap (.BMP), an executable (.EXE), or a dynamic-link library (.DLL). If the file contains more than one icon resource, specify the resource with a number after the filename—for example, My File.exe, 1. The numbering follows the same conventions as the registry. The default path for the file will be relative to the Autorun.inf file. If you want to specify an absolute path for an icon, use the following form.

```
defaulticon = path
```

The system automatically provides a pop-up menu for the icon and includes AutoPlay as the default command on that menu, so that double-clicking the icon will run the Open = line. You can include additional commands on the menu for the icon by adding entries for them in the Autorun.inf file, using the following form.

```
shell\verb\command = filename
shell\verb = Menu Item Name
```

To define an access key assignment for the command, precede the character with an ampersand (&). For example, to add the command Read Me First to the menu of the icon, include the following in Autorun.inf.

```
shell\readme\command = Notepad.exe My Directory\Readme.txt
shell\verb = Read &Me First
```

Although AutoPlay is typically the default menu item, you can define a different command to be the default by including the following line.

```
shell = verb
```

When the user double-clicks on the icon, the command associated with this entry will be executed.

System Naming Conventions

Windows provides support for filenames up to 255 characters long. Use the long filename when displaying the name of a file. Avoid displaying the filename extension unless the user chooses the option to display extensions or when the file type is not registered. The system automatically formats a filename correctly if you use the **SHGetFileInfo** or **GetFileTitle** function. For more information about these functions, see the documentation in the Microsoft Win32 Software Development Kit.

Because the system uses three-letter extensions to describe a file type, do not use extensions to distinguish different forms of the same file type. For example, if your application has a function that automatically backs up a file, name the backup file Backup of *filename.ext* (using its existing extension) or some reasonable equivalent, not *filename*.BAK. The latter implies a change of the file's type. Similarly, do not use a Windows filename extension unless your file fits the type description.

For more information about the common types recognized by the system, see Table 10.1, earlier in this chapter.

Long filenames can include any character, except

\ / : * ? < > |

When your application automatically supplies a filename, use a name that communicates information about its creation. For example, files created by a particular application should use either the application-supplied type name or the short type name as a proposed name—for example, worksheet or document. When that file exists already in the target directory, add a number to the end of the proposed name—for example, Document 2. When adding numbers to the end of a proposed filename, use the first number of an ordinal sequence that does not conflict with an existing name in that directory.

When you create a filename, the system automatically creates an MS-DOS filename (alias) for a file. The system displays both the long filename and the MS-DOS filename in the property sheet for the file.

When a file is copied, use the words "Copy of" as part of the generated filename—for example, "Copy of Sample" for a file named "Sample." If the prefix "Copy of" is already assigned to a file, the prefix includes a number—for example, "Copy 2 of Sample". You can apply the same naming scheme to links, except the prefix is "Link to" or "Shortcut to."

It is also important to support universal naming convention (UNC) paths for identifying the location of files and folders. UNC paths and filenames have the following form.

\\Server\Share\Directory\Filename.ext

Using UNC names enables the user to directly browse the network and open files without having to make explicit network connections.

Wherever possible, display the full name of a file. The number of characters you'll be able to display depends somewhat on the font used and the context in which the name is displayed. In any case, supply enough characters such that the user can reasonably distinguish between names. Take into account common prefixes such as "Copy of" or "Shortcut to".

You can also use an ellipsis to abbreviate a path name, in a displayable, but noneditable situation. In this case, include at least the first two entries of the beginning and the end of the path, using ellipses as notation for the names in between, as in the following example.

\\My Server\My Share\...\My Folder\My File

Taskbar Integration

The system provides support for integrating your application's interface with the taskbar. The following sections provide information on some of the capabilities and appropriate guidelines.

Taskbar Window Buttons

When an application creates a primary window, the system automatically adds a taskbar button for that window and removes it when that window closes. For some specialized types of applications that run in the background, a primary window may not be necessary. In such cases, make certain you provide reasonable support for controlling the application using the commands available on the application's icon; it should not appear as an entry in the taskbar, however.

The taskbar window buttons support drag and drop, but not in the conventional way. When the user drags an object over a taskbar window button, the system automatically restores the window. The user can then drop the object in the window.

Status Notification

The system allows you to add status or notification information to the taskbar (using the **Shell_NotifyIcon** function). Because the taskbar is a shared resource, add information to it that is of a global nature only or that needs monitoring by the user while working with other applications.

For more information about support for status notification and the **Shell_NotifyIcon** function, see the documentation in the Microsoft Win32 Software Developer's Kit.

Present status notification information in the form of a graphic supplied by your application, as shown in Figure 10.2.

Figure 10.2 Status indicator in the taskbar

When adding a status indicator to the taskbar, also support the following interactions.

- Provide a pop-up window that displays further information or controls for the object represented by the status indicator when the user clicks with button 1. If there is no information or control that applies, do not display anything. For example, the audio (speaker) status indicator displays a volume control. Use a pop-up window to supply for further information rather than a dialog box, because the user can dismiss the window by clicking elsewhere. Position the pop-up window near the status indicator so that the user can navigate to it quickly and easily.
- Display a pop-up menu for the object represented by the status indicator when the user clicks on the status indicator with button 2. On this menu, include commands that bring up property sheets or other windows related to the status indicator. For example, the audio status indicator provides commands that display the audio properties as well as the Volume Control mixer application. At a

minimum, include a What's This? context-sensitive Help command that displays information about the purpose of the status indicator.

- Execute the default command defined in the pop-up menu for the status indicator when the user double-clicks.
- Display a tooltip that indicates what the status indicator represents. For example, this could include the name of the indicator, a value, or both.

Message Notification

When your application's window is inactive but must display a message, rather than displaying a message box on top of the currently active window and switching the input focus, flash your application's taskbar window button to notify the user of the pending message. This avoids interfering with the user's current activity but lets the user know a message is waiting. When the user activates your application's window, the application can display a message box. Rather than flashing the button continually, flash the window button only a limited number of times (for example, three), then leave the button in the highlighted state, as shown in Figure 10.3. This lets the user know there is still a pending message.

Figure 10.3 A taskbar button used to notify a user of a pending message

This cooperative means of notification is preferable unless a message relates to system integrity or the user's data, in which case your application may immediately display a system modal message box. In such cases, flush the input queue so that the user does not inadvertently select a choice in that message box.

For more information about message boxes, see Chapter 8, "Secondary Windows."

Recycle Bin Integration

The Recycle Bin provides a repository for deleted files. If your application includes a facility for deleting files, support the Recycle Bin interface by using the **SHFileOperation** function. To support deletion to the Recycle Bin for other objects, first format the deleted data as a file by writing the it to a temporary file and then calling **SHFileOperation**. For more information about this function, see the documentation in the Microsoft Win32 Software Development Kit.

Control Panel Integration

The Windows Control Panel includes special objects that let users configure aspects of the system. Your application can add Control Panel objects or add property pages to the property sheets of existing Control Panel objects.

Adding Control Panel Objects

You can create your own Control Panel objects. Most Control Panel objects supply only a single secondary window, typically a property sheet. Define your Control Panel object to represent a concrete object rather than an abstract idea.

Every Control Panel object is a dynamic-link library. To ensure that the DLL can be automatically loaded by the system, set the file's extension to .CPL and install it in the Windows System directory.

Note The system automatically caches information about Control Panel objects in order to provide quick user access, provided that the Control Panel object supports the correct system interfaces. For more information about developing Control Panel objects, see the documentation in the Microsoft Win32 Software Development Kit.

Adding to the Passwords Object

The Passwords object in the Control Panel supplies a property sheet that allows the user to set security options and manage passwords for all password-protected services in the system. The Passwords object also allows you to add the name of a password-protected service to the object's list of services and use the Windows login password to be used as the password for all password-protected services in the system.

When you add your service to the Passwords object, the name of the service appears in the Select Password dialog box that appears when the user chooses Change Other Passwords. The user can then change the password for the service by selecting the name and filling in the resulting dialog box. The name of your service also appears in the Change Windows Password dialog box; the name appears with a check box next to it. By setting the check box option, the user chooses to keep the password for the service identical to the Windows login password. Similarly, the user can disassociate the service from the Windows login password by toggling the check box setting off.

To add your service to the Passwords object, register your service under the **HKEY_LOCAL_MACHINE** key.

```
HKEY_LOCAL_MACHINE
               System
                   CurrentControlSet
                       Control
                           PwdProvider
                               Provider Name
```

For more information about registering your password service, see the documentation in the Microsoft Win32 Software Development Kit.

You can also add a page to the property sheet of the Passwords object to support other security-related services that the user can set as property values. Add a property page if your application provides security-related functionality beyond simple activation and changing of passwords. To add a property page, follow the conventions for adding shell extensions.

For more information about supporting shell extensions that add property pages, see the "Shell Extensions" section earlier in this chapter.

Plug and Play Support

Plug and Play is a feature of Windows that, with little or no user intervention, automatically installs and configures drivers when their corresponding hardware peripherals are plugged into a PC. This feature applies to peripherals designed according to the Plug and Play specification. Supporting and appropriately adapting to Plug and Play hardware change can make your application easier to use. Following are some examples of supporting Plug and Play.

- Resizing your windows and toolbars relevant to screen size changes.
- Prompting users to shut down and save their data when the system issues a low power warning.
- Warning users about open network files when undocking their computers.
- Saving and closing files appropriately when users eject or remove removable media or storage devices or when network connections are broken.

System Settings and Notification

The system provides standard metrics and settings for user interface aspects, such as colors and fonts. The system also notifies running applications when its settings change (using the WM_SETTINGSCHANGE message). When

your application starts up, query the system to set your application's user interface to match the system parameters to ensure visual and operational consistency. Also, design your application to adjust itself appropriately when the system notifies it of changes to these settings.

For more information about supporting standard system settings, see the *Microsoft Win32 Programmer's Reference*.

Modeless Interaction

When designing your application, try to ensure that it is as interactive and nonmodal as possible. Here are some suggested ways of doing this:

- Use modeless secondary windows wherever possible.
- Segment processes, like printing, so you do not need load the entire application to perform the operation.
- Make long processes run in the background, keeping the foreground interactive. For example, when something is printing, it should be possible to minimize the window even if the document cannot be altered.

The multitasking support of Windows provides for defining separate processes, or *threads*, in the background. For more information about threads, see the documentation in the Microsoft Win32 Software Development Kit.

CHAPTER 11

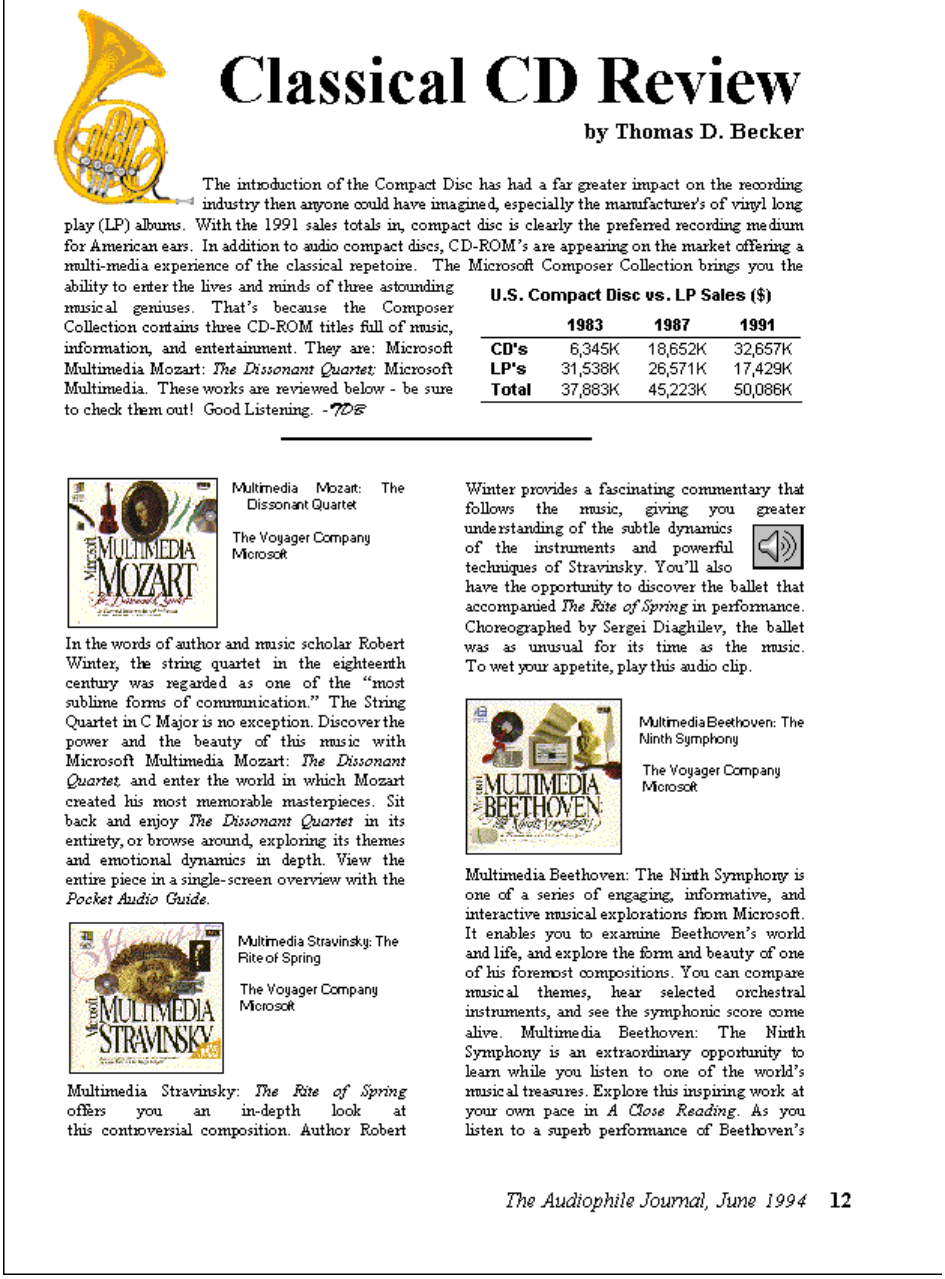
Working with OLE Embedded and OLE Linked Objects

Microsoft OLE is a standard set of system interfaces that enables users to combine objects supported by different applications. This chapter outlines guidelines for the interface for OLE embedded and OLE linked objects. While the primary focus assumes implementation using Microsoft OLE technology, you can apply many of these guidelines to the interaction between containers and their components.

The Interaction Model

As data becomes the major focus of interface design, its content is what occupies the user's attention, not the application managing it. In such a design, data is not limited to its native creation and editing environment; that is, the user is not limited to creating or editing data only within its associated application window. Instead, data can be transferred to other types of containers while maintaining its viewing and editing capability in the new container. Compound documents are a common example and illustration of the interaction between containers and their components, but they are not the only expression of this kind of object relationship that OLE can support.

Figure 11.1 shows an example of a compound document. The document includes word-processing text, tabular data from a spreadsheet, a sound recording, and pictures created in other applications.



Classical CD Review

by Thomas D. Becker

The introduction of the Compact Disc has had a far greater impact on the recording industry then anyone could have imagined, especially the manufacturer's of vinyl long play (LP) albums. With the 1991 sales totals in, compact disc is clearly the preferred recording medium for American ears. In addition to audio compact discs, CD-ROM's are appearing on the market offering a multi-media experience of the classical repetoire. The Microsoft Composer Collection brings you the ability to enter the lives and minds of three astounding musical geniuses. That's because the Composer Collection contains three CD-ROM titles full of music, information, and entertainment. They are: Microsoft Multimedia Mozart: *The Dissonant Quartet;* Microsoft Multimedia. These works are reviewed below - be sure to check them out! Good Listening. -*TDB*

U.S. Compact Disc vs. LP Sales ($)

	1983	1987	1991
CD's	6,345K	18,652K	32,657K
LP's	31,538K	26,571K	17,429K
Total	37,883K	45,223K	50,086K

Multimedia Mozart: The Dissonant Quartet

The Voyager Company
Microsoft

In the words of author and music scholar Robert Winter, the string quartet in the eighteenth century was regarded as one of the "most sublime forms of communication." The String Quartet in C Major is no exception. Discover the power and the beauty of this music with Microsoft Multimedia Mozart: *The Dissonant Quartet,* and enter the world in which Mozart created his most memorable masterpieces. Sit back and enjoy *The Dissonant Quartet* in its entirety, or browse around, exploring its themes and emotional dynamics in depth. View the entire piece in a single-screen overview with the *Pocket Audio Guide.*

Multimedia Stravinsky: The Rite of Spring

The Voyager Company
Microsoft

Multimedia Stravinsky: *The Rite of Spring* offers you an in-depth look at this controversial composition. Author Robert Winter provides a fascinating commentary that follows the music, giving you greater understanding of the subtle dynamics of the instruments and powerful techniques of Stravinsky. You'll also have the opportunity to discover the ballet that accompanied *The Rite of Spring* in performance. Choreographed by Sergei Diaghilev, the ballet was as unusual for its time as the music. To wet your appetite, play this audio clip.

Multimedia Beethoven: The Ninth Symphony

The Voyager Company
Microsoft

Multimedia Beethoven: The Ninth Symphony is one of a series of engaging, informative, and interactive musical explorations from Microsoft. It enables you to examine Beethoven's world and life, and explore the form and beauty of one of his foremost compositions. You can compare musical themes, hear selected orchestral instruments, and see the symphonic score come alive. Multimedia Beethoven: The Ninth Symphony is an extraordinary opportunity to learn while you listen to one of the world's musical treasures. Explore this inspiring work at your own pace in *A Close Reading.* As you listen to a superb performance of Beethoven's

The Audiophile Journal, June 1994 **12**

Figure 11.1 A compound document

How was this music review created? First, a user created a document and typed the text, then moved, copied, or linked content from other documents. Data objects that, when moved or copied, retain their native, full-featured editing and operating capabilities in their new container are called *OLE embedded objects*.

A user can also link information. An *OLE linked object* represents or provides access to another object that is in another location in the same container or in a different, separate container.

Generally, containers support any level of nested OLE embedded and linked objects. For example, a user can embed a chart in a worksheet, which, in turn, can be embedded within a word-processing document. The model for interaction is consistent at each level of nesting.

Creating OLE Embedded and OLE Linked Objects

OLE embedded and linked objects are the result of transferring existing objects or creating new objects of a particular type.

Transferring Objects

Transferring objects into a document follows basic command and direct manipulation interaction methods. The following sections provide additional guidelines for these commands when you use them to create OLE embedded or linked objects.

For more information about command and direct manipulation transfer methods, see Chapter 5, "General Interaction Techniques."

The Paste Command

As a general rule, using the Paste command should result in the most complete representation of a transferred object; that is, the object is embedded. However, containers that directly handle the transferred object can accept it optionally as native data instead of embedding it as a separate object, or as a partial or transformed form of the object if that is more appropriate for the destination container.

Use the format of the Paste command to indicate to the user how a transferred object is incorporated by a container. When the user copies a file object, if the container can embed the object, include the object's filename as a suffix to the Paste command. If the object is only a portion of a file, use the short type name—for example, Paste Worksheet or Paste Recording—as shown in Figure 11.2. A short type name can be derived from information stored in the registry. A Paste command with no name implies that the data will be pasted as native information.

For more information about type names, see Chapter 10, "Integrating with the System."

Figure 11.2 The Paste command with short type name

The Paste Special Command

Supply the Paste Special command to give the user explicit control over pasting in the data as native information, an OLE embedded object, or an OLE linked object. The Paste Special command displays its associated dialog box, as shown in Figure 11.3. This dialog box includes a list box with the possible formats that the data can assume in the destination container.

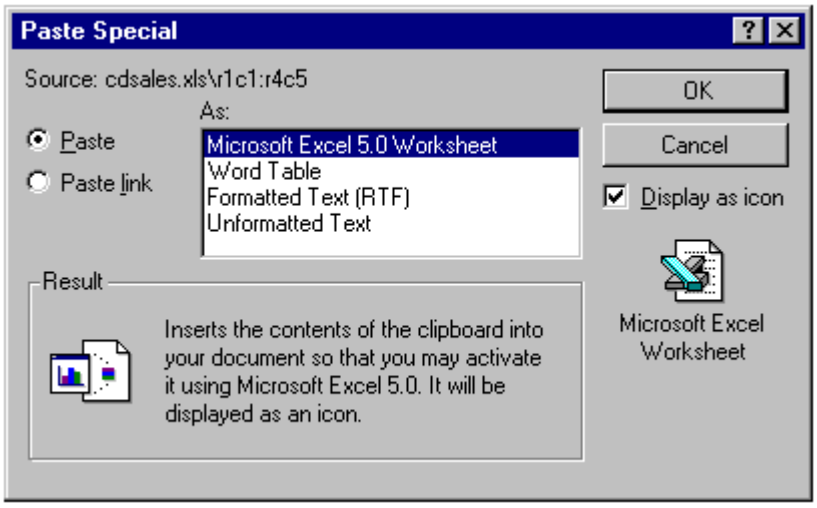


Figure 11.3 The Paste Special dialog box

Note The Microsoft Win32 Software Development Kit includes the Paste Special dialog box and other OLE-related dialog boxes that are described in this chapter.

In the formats listed in the Paste Special dialog box, include the object's full type name first, followed by other appropriate native data forms. When a linked object has been cut or copied, precede its object type by the word "Linked" in the format list. For example, if the user copies a linked Microsoft Excel 5.0 worksheet, the Paste Special

dialog box shows "Linked Microsoft Excel 5.0 Worksheet" in the list of format options because it inserts an exact duplicate of the original linked worksheet. Native data formats begin with the destination application's name and can be expressed in the same terms the destination identifies in its own menus. The initially selected format in the list corresponds to the format that the Paste Special command uses. For example, if the Paste Special command displays Paste *Object Filename* or Paste *Short Type Name* because the data to be embedded is a file or portion of a file, this is the format that is initially selected in the Paste Special list box.

To support creation of a linked object, the Paste Special dialog box includes a Paste link option. Figure 11.4 shows this option.

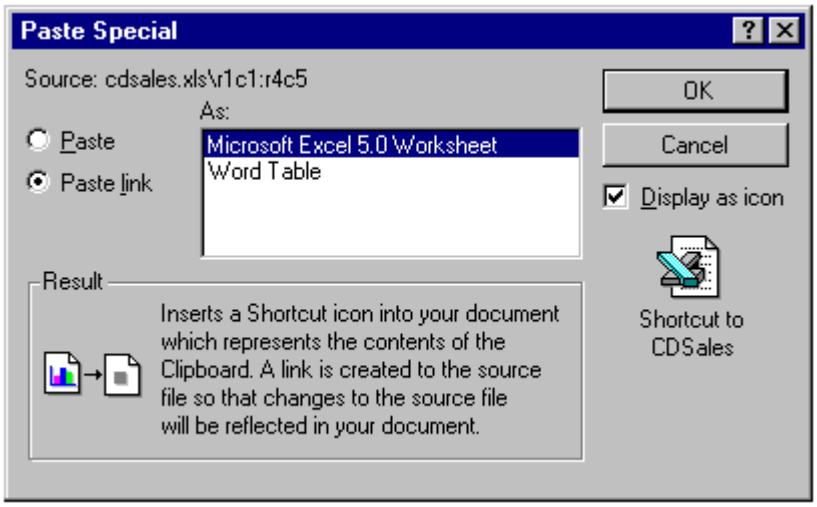


Figure 11.4 Paste Special dialog b with Paste link option set

A Display as Icon check box allows the user to choose displaying the OLE embedded or linked object as an icon. At the bottom of the dialog box is a section that includes text and pictures that describe the result of the operation. Table 11.1 lists the descriptive text for use in the Paste Special dialog box.

Table 11.1 Descriptive Text for Paste Special

Function	Descriptive text
Paste as an embedded object.	"Inserts the contents of the Clipboard into your document so you that you may activate it using *CompanyName ApplicationName*."
Paste as an embedded object so that it appears as an icon.	"Inserts the contents of the Clipboard into your document so you that you may activate it using *CompanyName ApplicationName application*. It will be displayed as an icon."
Paste as native data.	"Inserts the contents of the Clipboard into your document as *native type name [and optionally an additional Help sentence]*."
Paste as a linked object.	"Inserts a picture of the contents of the Clipboard into your document. Paste Link creates a link to the source file so that changes to the source file will be reflected in your document."
Paste as a linked object so that it appears as a shortcut icon.	"Inserts a Shortcut icon into your document which represents the contents of the Clipboard. A link is created to the source file so that changes to the source file will be reflected in your document."
Paste as linked native data.	"Inserts the contents of the Clipboard into your document as *native type name*. A link is created to the source file so that changes to the source file will be reflected in your document."

The Paste Link, Paste Shortcut, and Create Shortcut Commands

If linking is a common function in your application, you can optionally include a command that optimizes this process. Use Paste Link to support creating a linked object or linked native data. When using the command to create a linked object, include the name of the object preceded by the word "to"—for example, "Paste Link to Latest Sales." Omitting the name implies that the operation results in linked native data.

Use a Paste Shortcut command to support creation of a linked object that appears as a shortcut icon. You can also include a Create Shortcut command that creates a shortcut icon in the container. Apply these commands to containers where icons are commonly used.

Direct Manipulation

You can also support direct manipulation interaction techniques, such as drag and drop, for creating OLE embedded or linked objects. When the user drags a selection into a container, the container application can interpret the operation using information supplied by the source, such as the selection's type and format and by the destination container's own context, such as the container's type and its default transfer operation. For example, dragging a spreadsheet cell selection into a word-processing document can result in an OLE embedded table object. Dragging the same cell selection within the spreadsheet, however, would likely result in simply transferring the data in the cells.

Similarly, the destination container in which the user drops the selection also determines whether the dragged object creates an OLE linked object. For nondefault OLE drag and drop, the container application displays the appropriate transfer commands on the resulting pop-up menu and carries out the operation corresponding to the user's choice on that menu. The choices may include multiple commands that transfer the data in a different format or presentation. For example, a container application could offer the following choices for creating links: Link Here, Link *Short Type Name* Here, and Create Shortcut Here.

For more information about using direct manipulation for moving, copying, and linking objects, see Chapter 5, "General Interaction Techniques."

The default appearance of a transferred object also depends on the destination container application. For most types of documents, display the data or content presentation of the object (or in the case of an OLE linked object, a representation of the content), rather than as an icon. If the user chooses Create Shortcut Here as the transfer operation, the transferred object is displayed as an icon. If the object cannot be displayed as content—for example, because it does not support OLE—the object is displayed as an icon.

Inserting New Objects

In addition to transferring objects, you can support user creation of OLE embedded or linked objects by generating a new object based on an existing object or object type and inserting the new object into the target container.

The Insert Object Command

Include an Insert Object command on the menu responsible for creating or importing new objects into a container, such as an Insert menu. If no such menu exists, use the Edit menu. When the user selects this command, display the Insert Object dialog box, as shown in Figure 11.5. This dialog box allows the user to generate new objects based on their object type or an existing file.

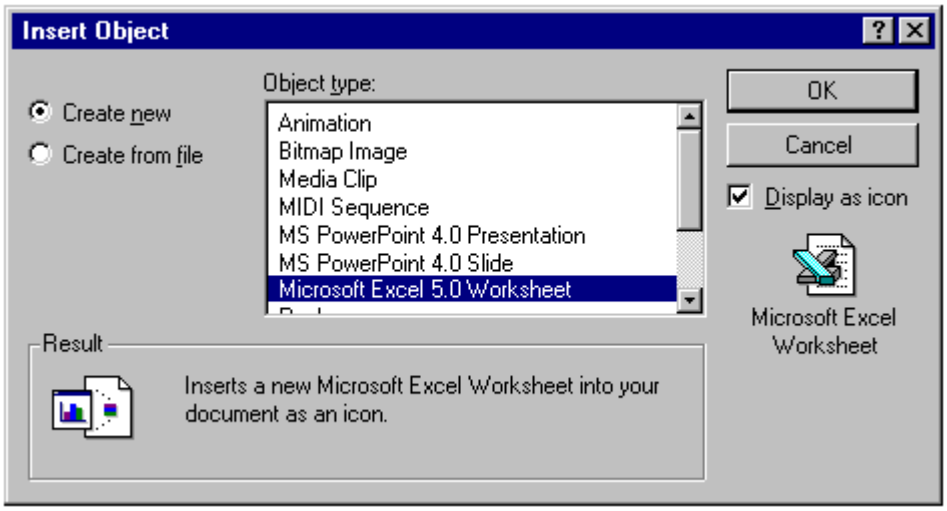


Figure 11.5 The Insert Object dialog box

The type list is composed of the type names of registered types. When the user selects a type from the list box and chooses the OK button, an object of the selected type is created and embedded.

For more information on type names and the registry, see Chapter 10, "Integrating with the System."

The user can also create an OLE embedded or linked object from an existing file, using the Create From File and Link options. When the user sets these options and chooses the OK button, the result is the same as directly copying or linking the selected file.

When the user chooses the Create From File option button, the Object Type list is removed, and a text box and Browse button appear in its place, as shown in Figure 11.6.

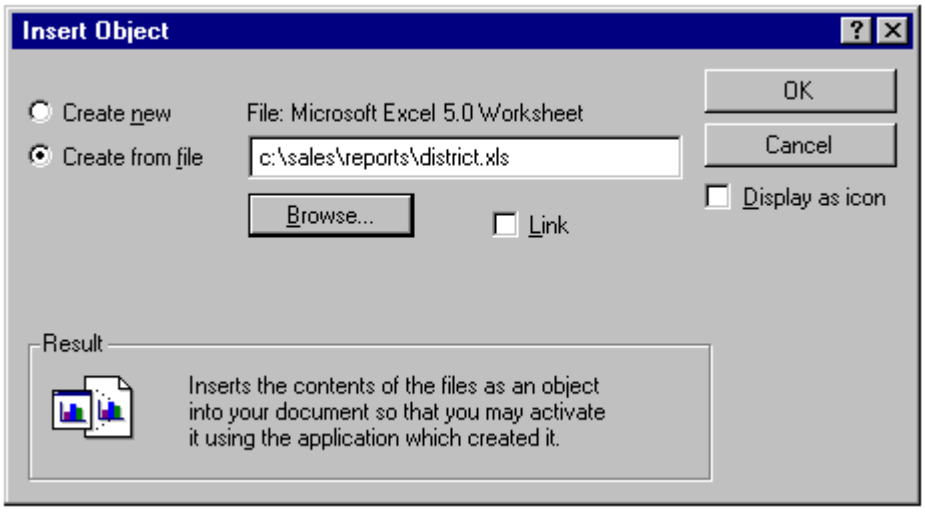


Figure 11.6 Creating an OLE embedded object from an existing file

The text box initially includes the current directory as the selection. The user can edit the current directory path when specifying a file. Use the file's type to determine the type of OLE embedded or linked object. Ignore any selection formerly displayed in the Object Type list box (shown in Figure 11.5).

Use the Link check box to support the creation of an OLE linked object to the file specified. The Insert Object dialog box displays this option only when the user chooses the Create From File option. This means that a user cannot insert an OLE linked object when choosing the Create New option button, because linked objects can be created only from existing files.

The Display As Icon check box in the Insert Object dialog box enables the user to specify whether to display the OLE embedded or linked object as an icon. When this option is set, the icon appears beneath the check box. An OLE linked object displayed as an icon is the equivalent of a shortcut icon.

Note If the user chooses a non-OLE file for insertion, it can be inserted only as an icon. The result is an OLE package. A *package* is an OLE encapsulation of a file so that it can be embedded in an OLE container. Because packages support limited editing and viewing capabilities, support OLE for all your object types so they will not be converted into packages.

At the bottom of the Insert Object dialog box, text and pictures describe the final outcome of the insertion. Table 11.2 outlines the syntax of descriptive text to use within the Insert Object dialog box.

Table 11.2 Descriptive Text for Insert Object Dialog Box

Function	Resulting text
Creates a new (embedded) object from a selected type.	"Inserts a new *Type Name* into your document."
Creates a new object from a selected type and displays it as an icon.	"Inserts a new *Type Name* into your document as an icon."
Creates a new object based on a selected file.	"Inserts the contents of the file as an object into your document so that you may activate it using the application which created it."
Creates a new object based on a selected file and displays it as an icon.	"Inserts the contents of the file as an object into your document so that you may activate it using the application which created it. It will be displayed as an icon."
Creates a new object that is linked to a selected file.	"Inserts a picture of the file contents into your document. The picture will be linked to the file so that changes to the file will be reflected in your document."
Creates a new object that is linked to a selected file and displays it as a Shortcut icon.	"Inserts a Shortcut icon into your document which represents the file. The Shortcut icon will be linked to the original file, so that you can quickly open the original from inside your document."

You can also use the context of the current selection in the container to determine the format of the newly created object and the effect of it being inserted into the container. For example, an inserted graph can automatically reflect the data in a selected table. Use the following guidelines to support predictable insertion:

- If an inserted object is not based on the current selection, follow the same conventions as for a Paste command and add or replace the selection depending on the context. For example, in text or list contexts, where the selection represents a specific insertion location, replace the active selection. For nonordered or Z-ordered contexts, where the selection does not represent an explicit insertion point, add the object, using the destination context to determine where to place the object.

For more information about the guidelines for inserting an object with a Paste command, see Chapter 5, "General Interaction Techniques."

- If the new object is automatically connected (linked) to the selection (for example, table data and a graph), insert the new object in addition to the selection and make the inserted object the new selection.

- If the object is based on, but does not remain connected to, the selection, the new object's application (not the container application) determines whether to remove the given selection. In this case, consider the specific use of an object type to determine whether adding the object or replacing the current selection is more meaningful.

After inserting an OLE embedded object, activate it for editing. If you insert an OLE linked object, do not activate the object.

Other Techniques for Inserting Objects

The Insert Object command provides support for inserting all registered OLE objects. You can include additional commands tailored to provide access to common or frequently used object types. You can implement these as additional menu commands or as toolbar buttons or other controls. These buttons provide the same functionality as the Insert Object dialog box, but perform more efficiently. Figure 11.7 illustrates two examples. The drawing button inserts a new blank drawing object; the graph button creates a new graph that uses the data values from a currently selected table.

Figure 11.7 Drawing and graph buttons

Displaying Objects

When displaying an OLE embedded or linked object in its presentation or content form (as opposed to displaying the object as an icon), use a cached metafile description (though, it is possible for an object to draw itself directly in its container). In this presentation, the object may be visually indistinguishable from native objects, as shown in Figure 11.8.

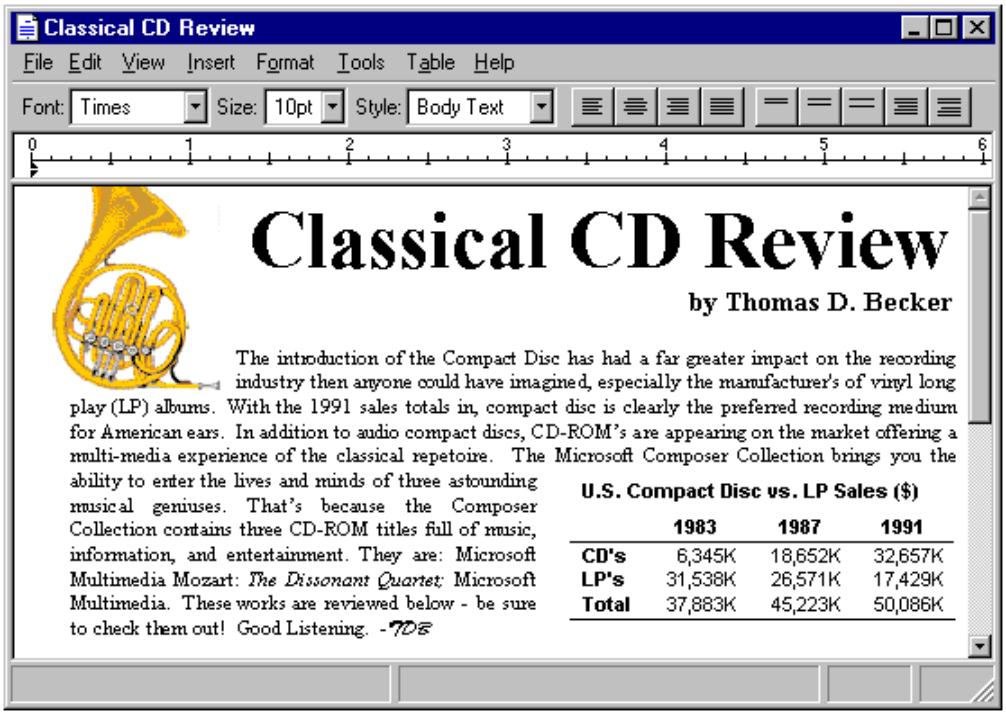


Figure 11.8 Compound document with indistinguishable objects

You may find it preferable to enable the user to visually identify OLE embedded or linked objects without interacting with them. To do so, provide a Show Objects command that, when chosen, displays a solid border, one pixel wide, drawn in the window text color (COLOR_WINDOWTEXT) around the extent of an OLE embedded object and a dotted border around OLE linked objects (shown in Figure 11.9). If the container application cannot guarantee that an OLE linked object is up-to-date with its source because of an unsuccessful automatic update or a manual link, the system should draw a dotted border using the system grayed text color (COLOR_GRAYTEXT) to suggest that the OLE linked object is out of date. The border should be drawn around a container's first-level objects only, not objects nested below this level.

For more information about COLOR_WINDOWTEXT and COLOR_GRAYTEXT and the **GetSysColor** function, see the *Microsoft Win32 Programmer's Reference*.

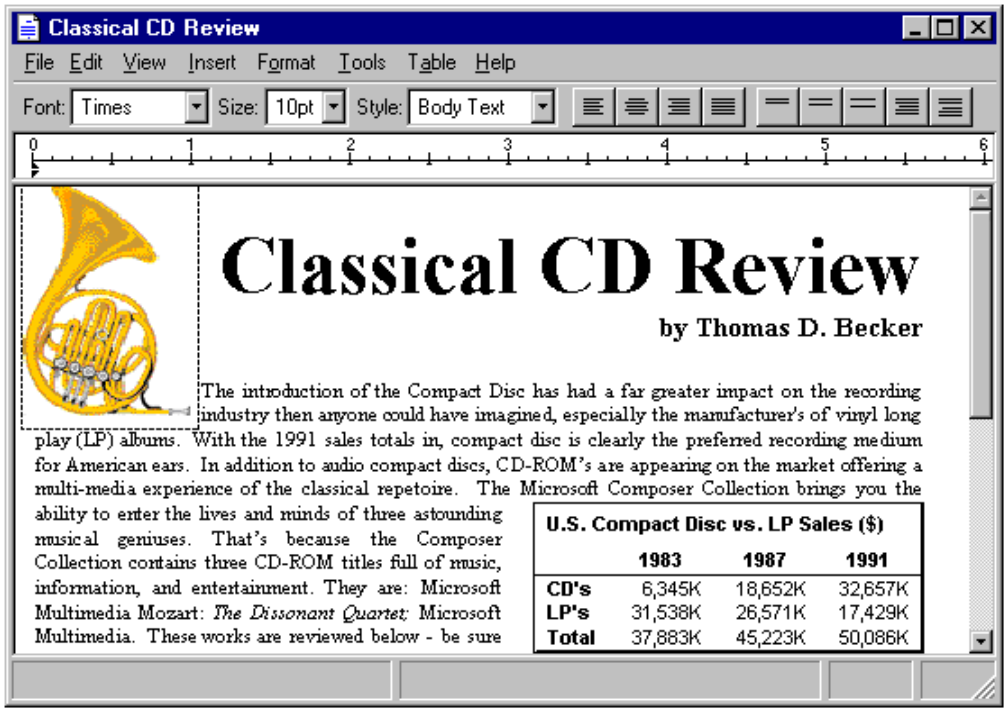


Figure 11.9 Compound document with distinguishable objects

If these border conventions are not adequate to distinguish OLE embedded and linked objects, you can optionally include additional distinctions; however, make them clearly distinct from the appearance for any standard visual states, and distinguish OLE embedded from OLE linked objects.

Whenever the user creates an OLE linked or embedded object with the Display As Icon check box set, display the icon using its original appearance, unless the user explicitly changes it. A linked icon also includes the shortcut graphic. If an icon is not registered in the registry for the object, use the system-generated icon.

An icon includes a label. When the user creates an OLE embedded object, define the icon's label to be one of the following, based on availability:

- The name of the object, if the object has an existing human-readable name such as a filename without its extension.
- The object's registered short type name (for example, Picture, Worksheet, and so on), if the object does not have a name.
- The object's registered full type name (for example, Microsoft Paint 1.0 Picture, Microsoft Excel 5.0 Worksheet), if the object has no name or registered short type name.
- "Document" if an object has no name, short type name, or registered type name.

When an OLE linked object is displayed as an icon, define the label using the source filename as it appears in the registry, preceded by the words "Shortcut to"—for example, "Shortcut to Annual Report." The path of the source is not included. Avoid displaying the filename extension unless the user chooses the system option to display extensions or the file type is not registered. The system automatically formats a filename correctly if the **SHGetFileInfo** function is used.

For more information about the **SHGetFileInfo** function, see the documentation included in the Microsoft Win32 Software Development Kit.

Follow the same conventions for labeling the shortcut icon when you create an OLE linked object to only a portion of a document (file). Because a container can include multiple links to different portions of the same file, optionally you may provide further identification to differentiate linked objects by appending a portion of the end of the link path (moniker). For example, you may want to include everything from the end of the path up to the last or next to last occurrence of a link path delimiter. The standard OLE link path delimiter that a link source should provide for identifying a data range is the exclamation character. However, the link path may include other types of delimiters. Be careful when deriving an identifier from the link path to format the additional information using only valid filename characters. Make sure that if the user transfers the shortcut icon to a folder or the desktop, the name can be used.

Selecting Objects

An OLE embedded or linked object follows the selection behavior and appearance techniques supported by its container; the container application supplies the specific appearance of the object. For example, Figure 11.10 shows how the linked drawing of a horn is handled as part of a contiguous selection in the document.

For more information about selection techniques, see Chapter 5, "General Interaction Techniques." For more information about selection appearance, see Chapter 13, "Visual Design."

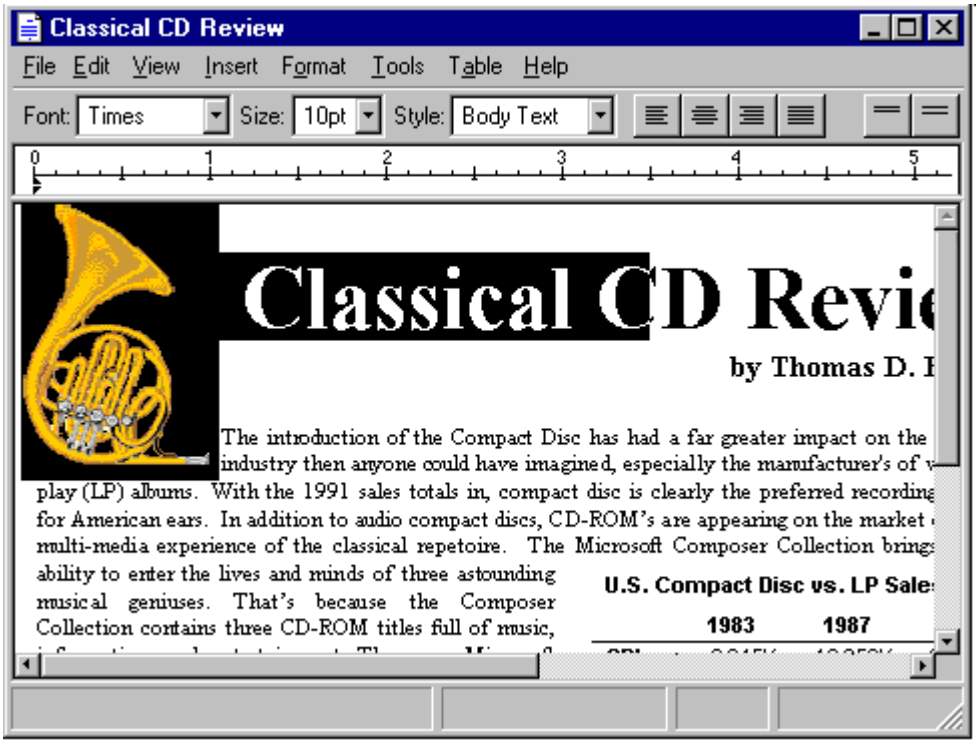


Figure 11.10 An OLE linked object as part of a multiple selection

When the user individually selects the object, display the object with an appropriate selection appearance for that type, as shown in Figure 11.11. For example, for the content view of an object, display it with handles. In addition, if your application's window includes a status bar that displays messages, display an appropriate description of how to activate the object. When the object is displayed as an icon, use the checkerboard selection highlighting used for icons in folders and on the desktop. For OLE linked objects, overlay the content view's lower left corner with the shortcut graphic.

For more information about status line messages, see Table 11.3, "File Menu Status Line Messages," later in this chapter.

Figure 11.11 An individually selected OLE linked object

Accessing Commands for Selected Objects

A container application always displays the commands that can be applied to its objects. When the user selects an OLE embedded or linked object as part of the selection of native data in a container, enable commands that apply to the selection as a whole. When the user individually selects the object, enable only commands that apply specifically to the object. Your application retrieves these commands from what has been registered by the object's type in the registry and displays these commands in the menus that are supplied for the object. If your application includes a menu bar, include the selected object's commands on a submenu of the Edit menu, or as a separate menu on the menu bar. Use the name of the object as the text for the menu item. If you use the short type name as the name of the object, add the word "Object." For an OLE linked object, use the short type name, preceded by the word "Linked." Figure 11.12 shows these variations.

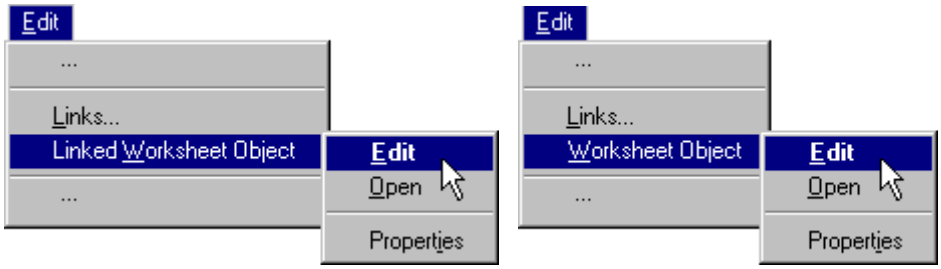


Figure 11.12 Selected object menus

Define the first letter of the word “Object”, or its localized equivalent, as the access character for keyboard users. When no object is selected, display the command with just the text, “Object”, and disable it.

A container application also provides a pop-up menu for a selected OLE object (shown in Figure 11.13), displayed using the standard interaction techniques for pop-up menus (clicking with mouse button 2). Include on this menu the commands that apply to the object as a whole as a unit of content, such as transfer commands, and the object's registered commands. In the pop-up menu, display the object's registered commands as individual menu items rather than in a cascading menu. It is not necessary to include the object's name or the word “Object” as part of the menu item text. In addition, provide a Properties command on the menu and on the property sheet of the selected object.

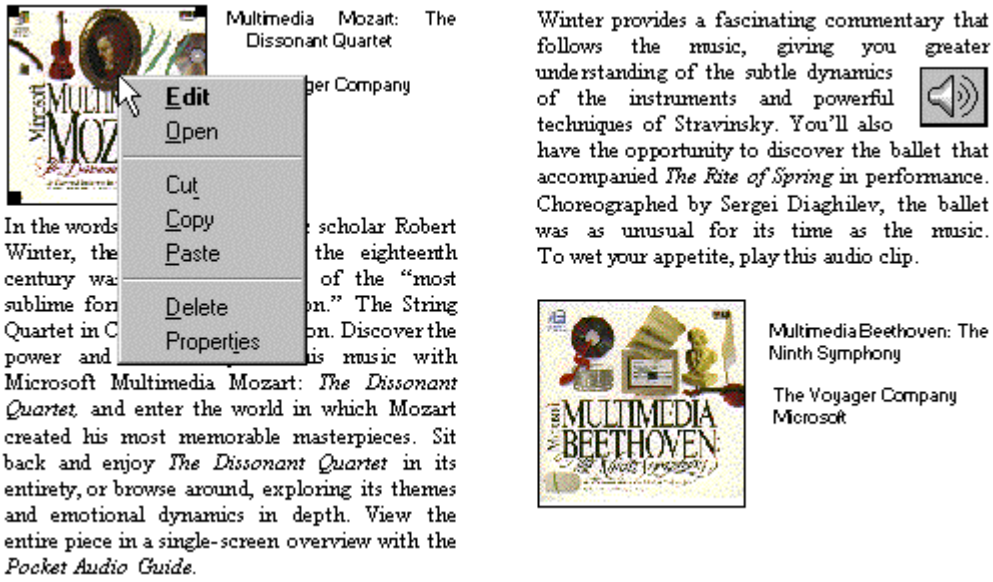


Figure 11.13 Pop-up menu for an OLE embedded picture

You can enable commands that depend on the state of the object. For example, a media object that uses Play and Rewind as operations disables Rewind when the object is at the beginning of the media object.

Note You can also support operations based on the selection appearance. For example, you can support operations such as resizing using the handles you supply. When the user resizes a selected OLE object, however, scale the presentation of the object, because there is no method by which another operation, such as cropping, can be applied to the OLE object.

If an object's type is not registered, you still supply any commands that can be appropriately applied to the object as content, such as a transfer command, alignment commands, and a Properties command. Also add an Open With command and define it as the default command for the object so that double-clicking starts the command. When the user chooses the Open With command, a system-supplied dialog box is displayed that enables the user to choose from a list of applications that can operate on the type or convert the object's type.

Activating Objects

While selecting an object provides access to commands applicable to the object as a whole, it does not provide access for the user to interact with the data or content of an object. Activating the object allows user interaction with the internal content of the object. There are two basic models for activating objects: outside-in activation and inside-out activation.

Outside-in Activation

Outside-in activation occurs when an activation command is carried out. Selecting an object that is already selected simply reselects that object and does not constitute an explicit action. The user activates the object by using a particular command such as Edit or Play, usually the object's command. Shortcut actions that correspond to these commands, such as double-clicking or pressing a shortcut key, can also activate the object. Most OLE container applications employ this model because it allows the user to easily select objects and reduces the risk of inadvertently activating an object whose underlying code may take a significant amount of time to load and dismiss.

When supporting outside-in activation, display the standard pointer (northwest arrow) over an outside-in activated object within your container when the object is selected but inactive. This indicates to the user that the outside-in object behaves as a single, opaque object. When the user activates the object, the object's application displays the appropriate pointer for its content. Use the registry to determine the object's activation commands.

Inside-out Activation

With *inside-out activation*, interaction with an object is direct; that is, the object is activated as the user moves the pointer over the extent of the object. From the user's perspective, inside-out objects are indistinguishable from native data, because the content of the object is directly interactive and no additional action is necessary. Use this method for the design of objects that benefit from direct interaction, or when activating the object has little effect on performance or use of system resources.

Inside-out activation requires closer cooperation between the container and the object. For example, when the user begins a selection within an inside-out object, the container must clear its own selection so that the behavior is consistent with normal selection interaction. An object supporting inside-out activation controls the appearance of the pointer as it moves over its extent and responds immediately to input. Therefore, to select the object as a whole, the user selects the border, or some other handle, provided by the object or its container. For example, the container application can support selection techniques such as region selection that select the object. Similarly, the object and its container can cooperate to support selection of the object. For example, if the user begins a selection by dragging within the object and crosses the extent of the object, the entire object becomes selected.

Although the default behavior for an OLE embedded object is outside-in activation, you can store information in the registry that indicates that an object's type (application class) is capable of inside-out activation (the

OLEMISC_INSIDEOUT constant) and prefers inside-out behavior (the OLEMISC_ACTIVATEWHENVISIBLE constant). You can set these values in a **MiscStatus** subkey, under the **CLSID** subkey of the **HKEY_CLASSES_ROOT** key. The values may be accessed using **IOleObject::GetMiscStatus**. The result of these values is subject to the container.

For more information about the registry, see Chapter 10, "Integrating with the System," and the *Microsoft OLE Programmer's Reference*.

Container Control of Activation

The container application determines how to activate its component objects: either it allows the inside-out objects to handle events directly or it intercedes and only activates them upon an explicit action. This is true regardless of the capability or preference of the object. That is, even though an object may register inside-out activation, it can be treated by a particular container as outside-in. Use an activation style for your container that is most appropriate for its specific use and is in keeping with its own native style of activation so that objects can be easily assimilated.

Regardless of the activation capability of the object, a container always consistently activates its content objects of the same type. Otherwise, the unpredictability of the interface is likely to impair its ease of use. Following are four potential container activation methods and when to use them.

Activation method	When to use
Outside-in throughout	This is the common design for containers that often embed large OLE objects and deal with them as whole units. Because many available OLE objects are not yet inside-out capable, most compound document editors support outside-in throughout to preserve uniformity.
Inside-out throughout	Ultimately, OLE containers will blend embedded objects with native data so seamlessly that the distinction dissolves. Inside-out throughout containers will become more feasible as increasing numbers of OLE objects support inside-out activation.
Outside-in plus inside-out preferred objects	Some containers may use an outside-in model for large, foreign embeddings but also include some inside-out preferred objects as though they were native objects (by supporting the OLEMISC_ACTIVATEWHENVISIBLE constant). For example, an OLE document might present form control objects as inside-out native data while activating larger spreadsheet and chart objects as outside-in.
Switch between inside-out throughout and outside-in throughout	Visual programming and forms layout design applications include design and run modes. In this type of environment, a container typically holds an object that is capable of inside-out activation (if not preferable) and alternates between outside-in throughout when designing and inside-out throughout when running.

OLE Visual Editing of OLE Embedded Objects

One of the most common uses for activating an object is editing its content in its current location. Supporting this type of in-place interaction is also called *OLE visual editing*, because the user can edit the object within the visual context of its container.

Unless the container and the object both support inside-out activation, the user activates an embedded object for visual editing by selecting the object and executing its Edit command, either from a drop-down or pop-up menu. You can also support shortcut techniques. For example, if Edit is the object's default operation, you can use double-clicking to activate the object for editing. Similarly, you can support pressing the ENTER key as a shortcut for activating the object.

Note Although earlier versions of OLE user interface documentation suggested using the ALT+ENTER key combination to activate an object if the ENTER key was already assigned, this key combination is now the recommended shortcut key for the Properties command. Instead, support the pop-up menu shortcut key, SHIFT+F10. This enables the user to activate the object by selecting the command from the pop-up menu.

When the user activates an OLE embedded object for OLE visual editing, the user interface for its content becomes available and blended into its container application's interface. The object can display its frame *adornments*, such as row or column headers, handles, or scroll bars, outside the extent of the object and temporarily cover neighboring material. The object's application can also change the menu interface, which can range from adding menu items to existing drop-down menus to replacing entire drop-down menus. The object can also add toolbars, status bars, and supplemental palette windows.

The degree of interface blending varies based on the nature of the OLE embedded object. Some OLE embedded objects may require extensive support and consequently result in dramatic changes to the container application's interface. Finer grain objects that emulate the native components of a container may have little or no need to make changes in the container's user interface. The container determines the degree to which an OLE embedded object's interface can be blended with its own, regardless of the capability or preference of the OLE embedded object. A container application that provides its own interface for an OLE embedded object can suppress an OLE embedded object's own interface. Figure 11.14 shows how the interface might appear when its embedded worksheet is active.

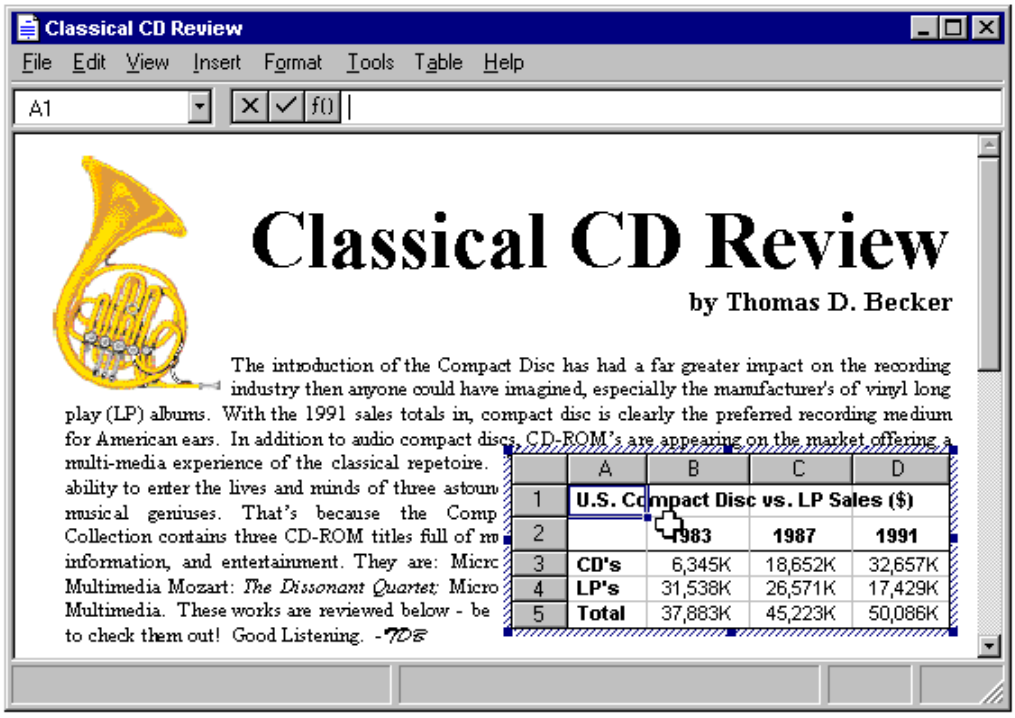


Figure 11.14 A worksheet activated for OLE visual editing

When the user activates an OLE embedded object, avoid changing the view and position of the rest of the content in the window. Although it may seem reasonable to scroll the window and thereby preserve the content's position, doing so can disturb the user's focus, because the active object shifts down to accommodate a new toolbar and shifts back up when it is deactivated. An exception may be when the activation exposes an area in which the container has nothing to display. Even in this situation, you may wish to render a visible region or filled area that corresponds to the background area outside the visible edge of the container so that activation keeps the presentation stable.

Activation does not affect the title bar. Always display the top-level container's name. For example, when the worksheet shown in Figure 11.14 is activated, the title bar continues to display the name of the document in which the worksheet is embedded and not the name of the worksheet. You can provide access to the name of the worksheet by supporting property sheets for your OLE embedded objects.

For more information about property sheets for OLE embedded objects, see the section, "Using Property Sheets," later in this chapter.

A container can contain multiply nested OLE embedded objects. However, only a single level is active at any one time. Figure 11.15 shows a document containing an active embedded worksheet with an embedded graph of its own. Clicking on the graph merely selects it as an object within the worksheet.

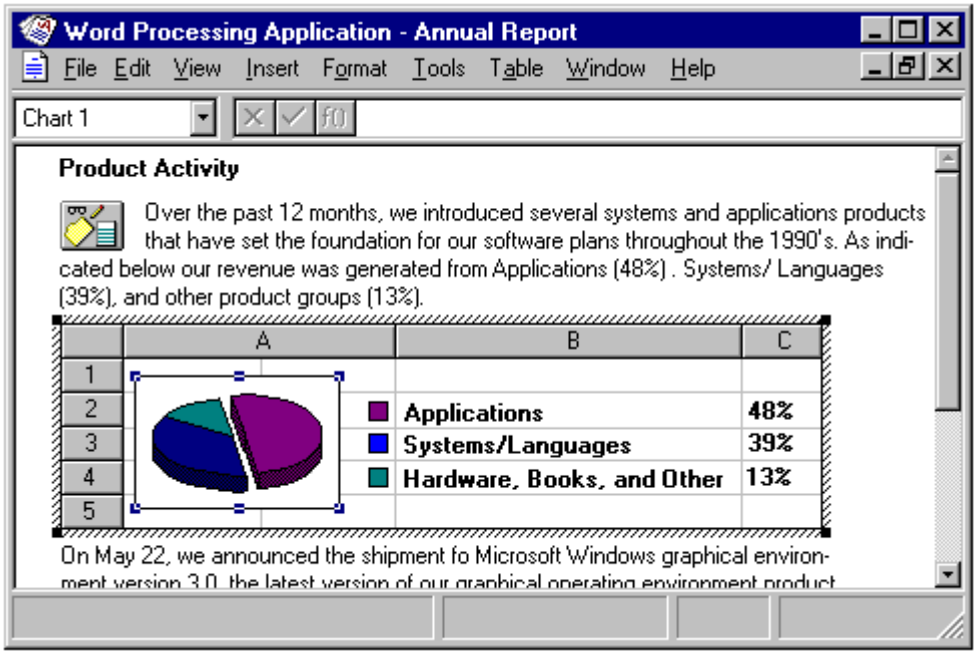


Figure 11.15 A selected graph within an active worksheet

Activating the embedded graph, for example, by choosing the graph's Edit command, activates the object for OLE visual editing, displaying the graph’s menus in the document's menu bar. This is shown in Figure 11.16. At any given time, only the interface for the currently active object and the topmost container are presented; intervening parent objects do not remain visibly active.

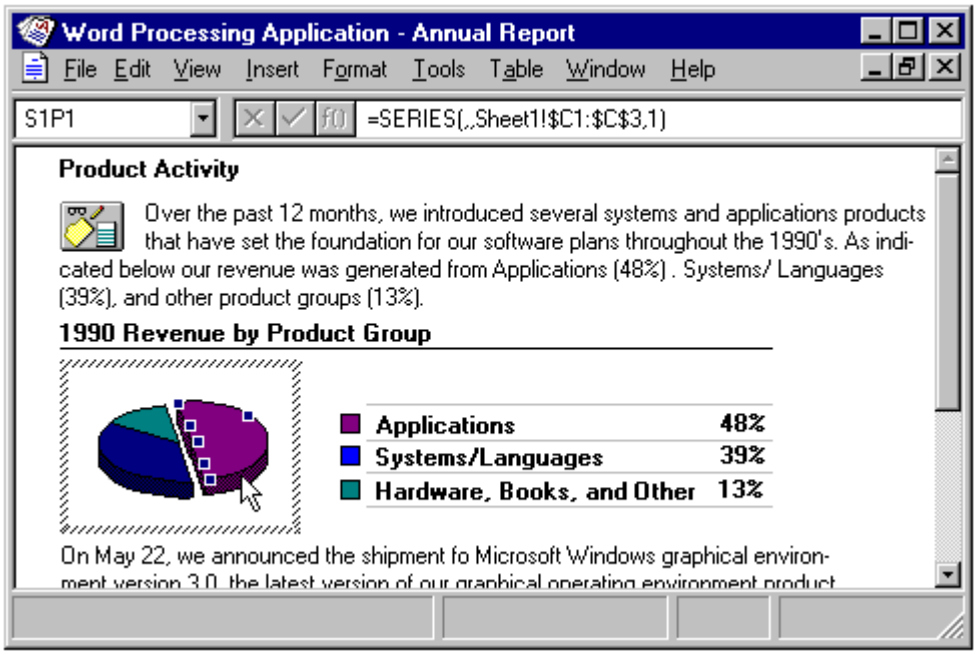


Figure 11.16 An active graph within a worksheet

Note Ideally, an OLE embedded object supports OLE visual editing on any scale because its container can be scaled arbitrarily. If an object cannot accommodate OLE visual editing in its container's current view scale, or if its container does not support OLE visual editing, open the object into a separate window for editing. For more information about OLE embedded objects, see the section, "Opening an Embedded Object," later in this chapter.

For any user interaction, such as when the user selects outside the extent of an active object or activates another object in the container, deactivate the current object and give the focus to the new object. This is also true for an object that is nested in the currently active object. An object application also supports user deactivation when the user presses the ESC key. If the object uses the ESC key at all times for its internal operation, the SHIFT+ESC key should deactivate the object, after which it becomes the selected object of its container.

Edits made to an active object become part of the container immediately and automatically, just like edits made to native data. Consequently, do not display an “Update changes?” message box when the object is deactivated. Remember that the user can abandon changes to the entire container, embedded or otherwise, if the topmost container includes an explicit command that prompts the user to save or discard changes to the container's file.

OLE embedded objects participate in the undo stack of the window in which they are activated.

For more information about embedded objects and the undo stack, see the section, "Undo for Active and Open Objects," later in this chapter.

While Edit is the most common command for activating an OLE embedded object for OLE visual editing, other commands can also create such activation. For example, when the user carries out a Play command on a video clip, you can display a set of commands that allow the user to control the clip (Rewind, Stop, and Fast Forward). In this case, the Play command provides a form of OLE visual editing.

The Active Hatched Border

If a container allows an OLE embedded object's user interface to change its user interface, then the active object's application displays a hatched border around itself to show the extent of the OLE visual editing context (shown in Figure 11.17). That is, if an active object places its menus in the topmost container's menu bar, display the active hatched border. The object's request to display its menus in the container's menu bar must be granted by the container application. If the object's menus do not appear in the menu bar (because the object did not require menus or the container refused its request for menu display), or the object is otherwise accommodated by the container's user interface, you need not display the hatched border.

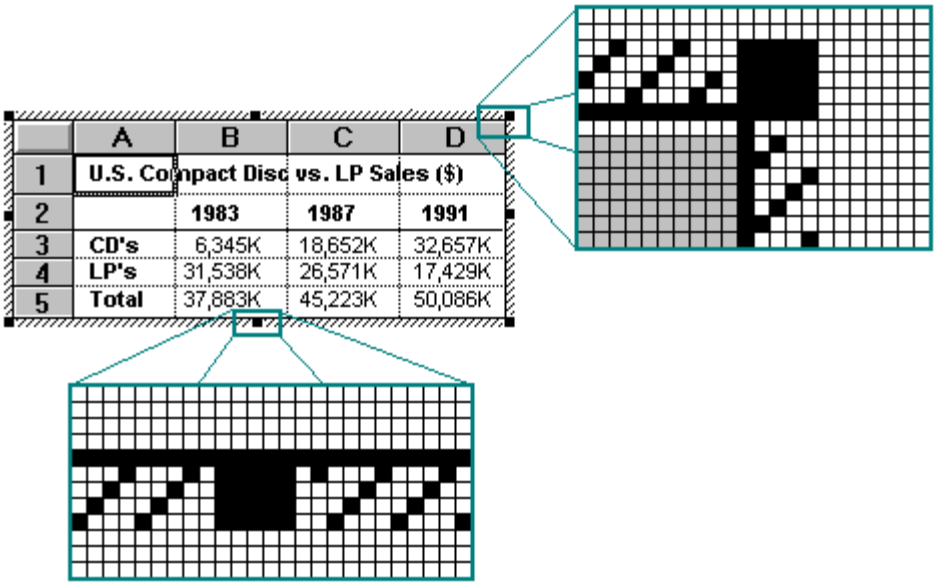


Figure 11.17 Hatched border around active OLE embedded objects

The hatched pattern is made up of 45-degree diagonal lines. The active object takes on the appearance that is best suited for its own editing; for example, the object may display frame adornments, table gridlines, handles, and other editing aids. Because the hatched border is part of the object's territory, the active object defines the pointer that appears when the user moves the mouse over the pointer.

Clicking in the hatched pattern (and not on the handles) is interpreted by the object as clicking just inside the edge of the border of the active object. The hatched area is effectively a hot zone that prevents inadvertent deactivations and makes it easier to select the content of the object.

Menu Integration

As the user activates different objects, different commands need to be accessed in the window's user interface. The following classification of menus—primary container menu, workspace menu, and active object menus—separates the interface based on menu groupings. This classification enhances the usability of the interface by defining the interface changes as the user activates or deactivates different objects.

Primary Container Menu

The topmost or primary container viewed in a primary window controls the work area of that window. If the primary container includes a menu bar, it supplies a single menu that includes commands that apply to the primary container as an entire unit. For document objects, use a File menu for this purpose, as shown in Figure 11.18. This menu includes document and file level commands such as Open, Save, and Print. Always display the primary container menu in the menu bar at all times, regardless of which object is active.

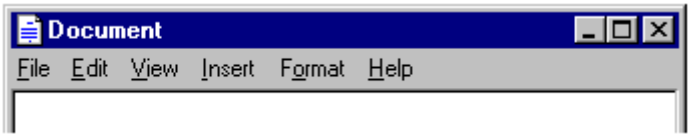


Figure 11.18 OLE visual editing menu layout

Workspace Menu

An MDI-style application also includes a workspace menu (typically labeled "Window") on its menu bar that provides commands for managing the document windows displayed within it, as shown in Figure 11.19. Like the primary container menu, the workspace menu should always be displayed, independent of object activation.

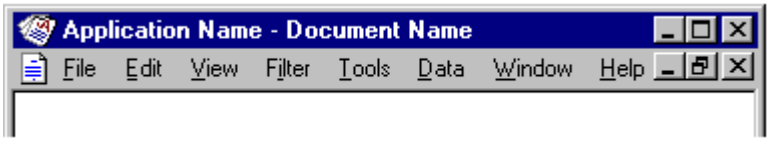


Figure 11.19 OLE visual editing menu layout for MDI

For more information about the multiple document interface (MDI), see Chapter 9, "Window Management."

Active Object Menus

Active objects can define menus that appear on the primary container's menu bar that operate on their content. Place commands for moving, deleting, searching and replacing, creating new items, applying tools, styles, and Help on these menus.

Active object commands apply only within the extent of the object. An active object's menus typically occupy the majority of the menu bar. Organize these menus following the same order and grouping that you display when the user opens the object into its own window. As their name suggests, active object commands are executed by the

currently active object. If no embedded objects are active, but the window is active, the primary container is the active object. Avoid naming your active object menus File or Window, because primary containers often use those titles. Objects that use direct manipulation as their sole interface need not provide active object menus or alter the menu bar when activated.

The active object can display a View menu. However, when the object is active, include only commands that apply to the object. If the object's container requires its document or window-level "viewing" commands to be available while an object is active, place them on a menu that represents the primary container window's pop-up menu and on the Window menu—if present.

When designing the interface of selected objects within an active object, follow the same guidelines as that of a primary container and one of its selected OLE embedded objects; that is, the active object displays the commands of the selected object (as registered in the registry) either as submenus of its menus or as separate menus.

An active object also has the responsibility for defining and displaying the pop-up menu for its content, specifically commands appropriate to apply to any selection within it. Figure 11.20 shows an example of a pop-up menu for a selection within an active bitmap image.

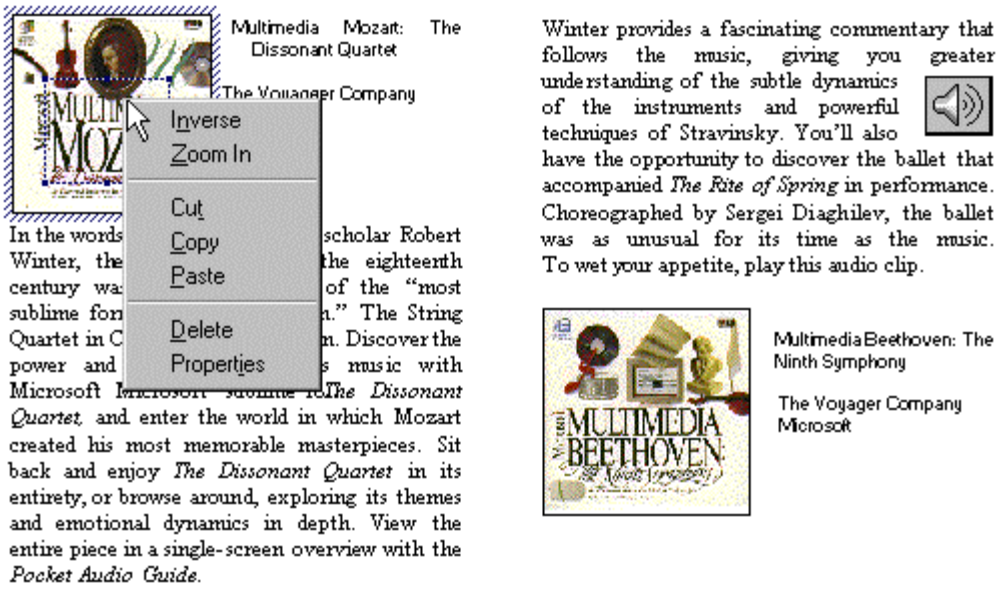


Figure 11.20 Active object pop-up menu

Keyboard Interface Integration

In addition to integrating the menus, you must also integrate the access keys and shortcut keys used in these menus.

Access Keys

The access keys assigned to the primary container's menu, an active object's menus, and MDI workspace menus should be unique. Following are guidelines for defining access keys for integrating these menu names.

- Use the first letter of the menu of the primary container as its access key character. Typically, this is "F" for File. Use "W" for a workspace's Window menu. Localized versions should use the appropriate equivalent.
- Use characters other than those assigned to the primary container and workspace menus for the menu titles of active OLE embedded objects. (If an OLE embedded object has previously existed as a standalone document, its application avoids these characters already.)
- Define unique access keys for an object's registered commands and avoid characters that are potential access keys for common container-supplied commands, such as Cut, Copy, Paste, Delete, and Properties.

Despite these guidelines, if the same access character is used more than once, pressing an ALT+*letter* combination cycles through the candidates, selecting the next match each time it is pressed. To carry out the command, the user must press the ENTER key when it is selected. This is standard system behavior for menus.

For more information about defining access keys, see Chapter 4, "Input Basics."

Shortcut Keys

For primary containers and active objects, follow the shortcut key guidelines covered in this guide. In addition, avoid defining shortcut keys for active objects that are likely to be assigned to the container. Include the standard editing and transfer (Cut, Copy, and Paste) shortcut keys, but avoid File menu or system-assigned shortcut keys. There is no provision for registering shortcut keys for a selected object's commands.

For more information about defining shortcut keys, see Chapter 4, "Input Basics," and Appendix B, "Keyboard Interface Summary."

If a container and an active object share a common shortcut key, the active object captures the event. That is, if the user activates an OLE embedded object, its application code directly processes the shortcut key. If the active object does not process the key event, it is available to the container, which has the option to process it or not. This applies to any level of nested OLE embedded objects. If there is duplication between shortcut keys, the user can always direct the key based on where the active focus is by activating that object. To direct a shortcut key to the container, the user deactivates an OLE embedded object—for example, by selecting in the container—but outside the OLE embedded object. Activation, not selection, of an OLE embedded object allows it to receive the keyboard events. The exception is inside-out activation, where activation results from selection.

Toolbars, Frame Adornments, and Palette Windows

Integrating drop-down and pop-up menus is straightforward because they are confined within a particular area and follow standard conventions. Toolbars, frame adornments (as shown in Figure 11.21), and palette windows can be constructed less predictably, so it is best to follow a replacement strategy when integrating these elements for active objects. That is, toolbars, frame adornments, and palette windows are displayed and removed as entire sets rather than integrated at the individual control level—just like menu titles on the menu bar.

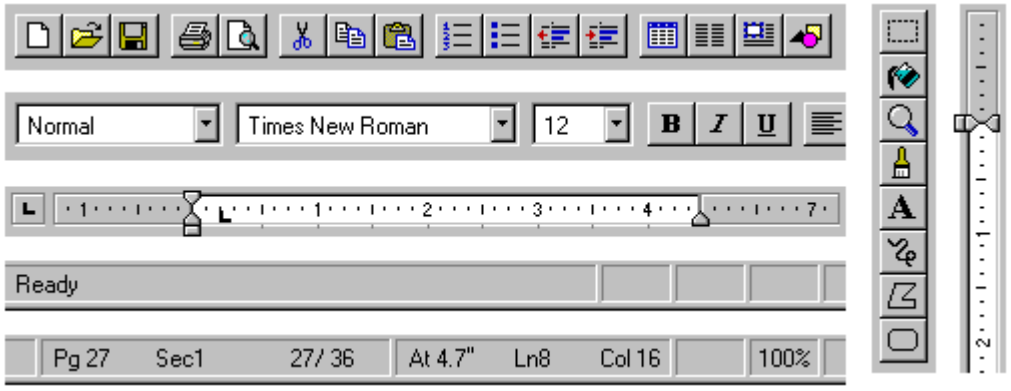


Figure 11.21 Examples of toolbars and frame adornments

When the user activates an object, the object application requests a specific area from its container in which to post its tools. The container application determines whether to:

- Replace its tool (or tools) with the tools of the object, if the requested space is already occupied by a container tool.
- Add the object's tool (or tools), if a container tool does not occupy the requested space.
- Refuse to display the tool (or tools) at all. This is the least desirable method.

Toolbars, frame adornments, and palette windows are all basically the same interfaces—they differ primarily in their location and the degree of shared control between container and object. There are four locations in the interface where these types of controls reside, and you determine their location by their scope.

Location	Description
Object frame	Places object-specific controls, such as a table header or a local coordinate ruler, directly adjacent to the object itself for tightly coupled interaction between the object and its interface. An object (such as a spreadsheet) can include scrollbars if its content extends beyond the boundaries of its frame.
Pane frame	Locates view-specific controls at the pane level. Rulers and viewing tools are common examples.
Document (primary container) window frame	Attaches tools that apply to the entire document (or documents in the case of an MDI window) just inside any edge of its primary window frame. Popular examples include ribbons, drawing tools, and status lines.
Windowed	Displays tools in a palette window—this allows the user to place them as desired. A palette window typically floats above the primary window and any other windows of which it is part.

For more information about the behavior of palette windows, see Chapter 8, "Secondary Windows."

When determining where to locate a tool area, avoid situations that cause the view to shift up and down as different-sized tool areas are displayed or removed as the user activates different objects. This can be disruptive to the user's task. Figure 11.22 shows possible positions for interface controls.

Figure 11.22 Possible positions for interface controls

Because container tool areas can remain visible while an object is active, they are available to the user simply by interacting with them—this can reactivate the container application. The container determines whether to activate or leave the object active. If toolbar buttons of an active object represent a primary container or workspace, commands, such as Save, Print, or Open, disable them.

For more information about the negotiation protocols used for activation, see the *Microsoft OLE Programmer's Reference*.

As the user resizes or scrolls its container's area, an active object and its toolbar or frame adornments placed on the object frame are clipped, as is all container content. These interface control areas lie in the same plane as the object.

Even when the object is clipped, the user can still edit the visible part of the object in place and while the visible frame adornments are operational.

Some container applications scroll at certain increments that may prevent portions of an OLE embedded object from being visually edited. For example, consider a large picture embedded in a worksheet cell. The worksheet scrolls vertically in complete row increments; the top of the pane is always aligned with the top edge of a row. If the embedded picture is too large to fit within the pane at one time, its bottom portion is clipped and consequently never viewed or edited in place. In cases like this, the user can open the picture into its own window for editing.

Window panes clip frame adornments of nested embedded objects, but not by the extent of any parent object. Objects at the very edge of their container's extent or boundary potentially display adornments that extend beyond the bounds of the container's defined area. In this case, if the container displays items that extend beyond the edge, display all the adornments; otherwise, clip the adornments at the edge of the container. Do not temporarily move the object within its container just to accommodate the appearance of an active embedded object's adornments. A pane-level control can potentially be clipped by the primary (or parent, in the case of MDI) window frame, and a primary window adornment or control is clipped by other primary windows.

Opening OLE Embedded Objects

The previous sections have focused on OLE visual editing—editing an OLE embedded object in place; that is, its current location is within its container. Alternatively, the user can also open embedded objects into their own window. This gives the user the opportunity of seeing more of the object or seeing the object in a different view state. Support this operation by registering an Open command for the object. When the user chooses the Open command of an object, it opens it into a separate window for editing, as shown in Figure 11.23.

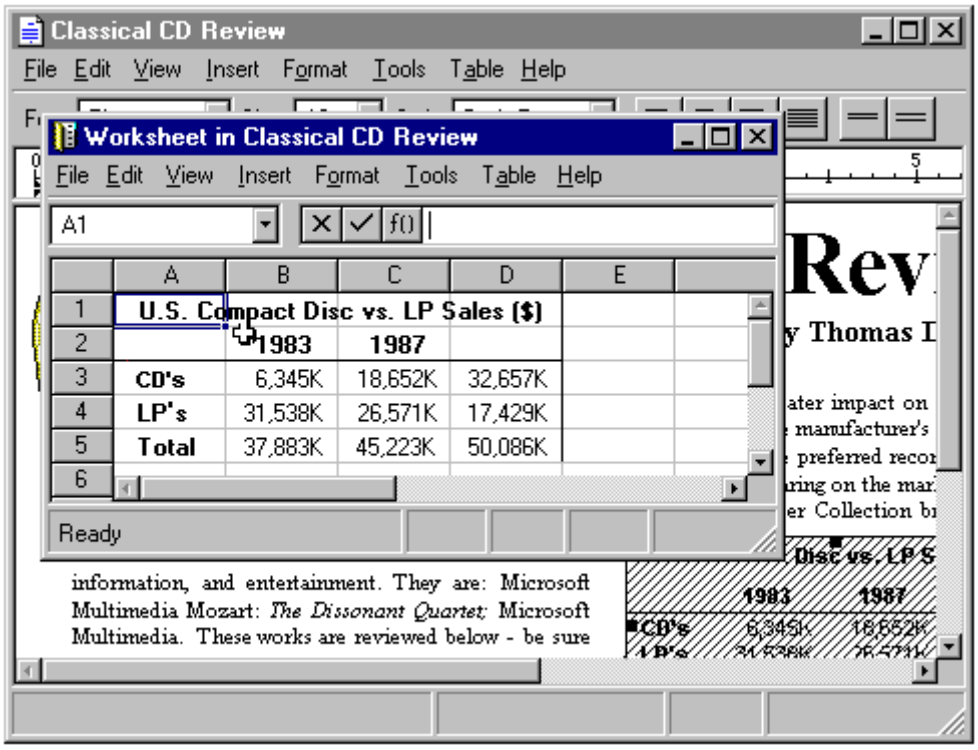


Figure 11.23 An opened OLE embedded worksheet

After opening an object, the container displays it masked with an "open" hatched (lines at a 45-degree angle) pattern that indicates the object is open in another window, as shown in Figure 11.24.

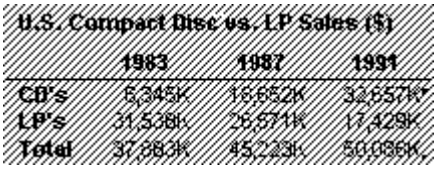


Figure 11.24 An opened object

Format the title text for the open object's window as "*Object Name* in *Container Name*" (for example, "Sales Worksheet in Classical CD Review"). Including the container's name emphasizes that the object in the container and the object in the open window are considered the same object.

Note This convention for the title bar text applies only when the user opens an embedded object. It does not apply when the user activates the object in place. In this latter case, do not change the title bar text.

An open OLE embedded object represents an alternate window onto the same object within the container as opposed to a separate application that updates changes to the container document. Therefore, edits are immediately and automatically reflected in the object in the document, and there is no longer the need for displaying an update confirmation message upon exiting the open window.

Nevertheless, you can still include an Update *Source File* command in the window of the open objects to allow the user to request an update explicitly. This is useful if you cannot support frequent “real-time” image updates because of operational performance. In addition, when the user closes an open object's window, automatically update its presentation in the container's window.

You may also include Import File and similar commands in the window of the open object. Treat importing a file into the window of the open embedded object the same as any change to the object.

When the user opens an object, it is the selected object in the container; however, the user can change the selection in the container afterwards. Like any selected OLE embedded object, the container supplies the appropriate selection appearance together with the open appearance, as shown in Figure 11.25. The selected and open appearances apply only to the object's appearance on the display. If the user chooses to print the container while an OLE embedded object is open or active, use the presentation form of objects; neither the open nor active hatched pattern should appear in the printed document because neither pattern is part of the content.

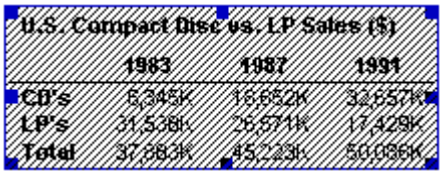


Figure 11.25 A selected open object

While an OLE embedded object is open, it is still a functioning member of its container. It can still be selected or unselected, and can respond to appropriate container commands. At any time, the user may open any number of OLE embedded objects. When the user closes its container window, deactivate and close the windows for any open OLE embedded objects.

When the user opens an OLE embedded object, if it has file operations, such as Open, remove these in the resulting window or replace them with commands such as Import to avoid severing the object's connection with its container. The objective is to present a consistent conceptual model; the object in the opened window is the same as the one in the container.

Editing an OLE Linked Object

An OLE linked object can be stored in a particular location, moved or copied, and has its own properties. Container actions can be applied in as much as the OLE linked object acts as a unit of content. So an OLE container supplies commands, such as Cut, Copy, Delete, and Properties, and interface elements such as handles, drop-down and pop-up menu items, and property sheets, for the OLE linked objects it contains.

For more information about providing access to selected OLE objects, see the section, "Accessing Commands for Selected Objects," earlier in this chapter.

The container also provides access to the commands that activate the OLE linked object, including the commands that provide access to content represented by the OLE linked object. These commands are the same as those that have been registered for the link source's type. Because an OLE linked object represents and provides access to another object that resides elsewhere, editing an OLE linked object always takes the user back to the link source. Therefore, the command used to edit an OLE linked object is the same as the command of its linked source object. For example, the menu of a linked object can include both Open and Edit if its link source is an OLE embedded object. The Open command opens the embedded object, just as carrying out the command on the OLE embedded object does. The Edit command opens the container window of the OLE embedded object and activates the object for OLE visual editing.

Figure 11.26 shows the result of opening a linked bitmap image of a horn. The image appears in its own window for editing. Note that changes made to the horn are reflected not only in its host container, the "Classical CD Review" document, but in every other document that contains an OLE linked object linked to that same portion of the "Horns" document. This illustrates both the power and the potential danger of using links in documents.

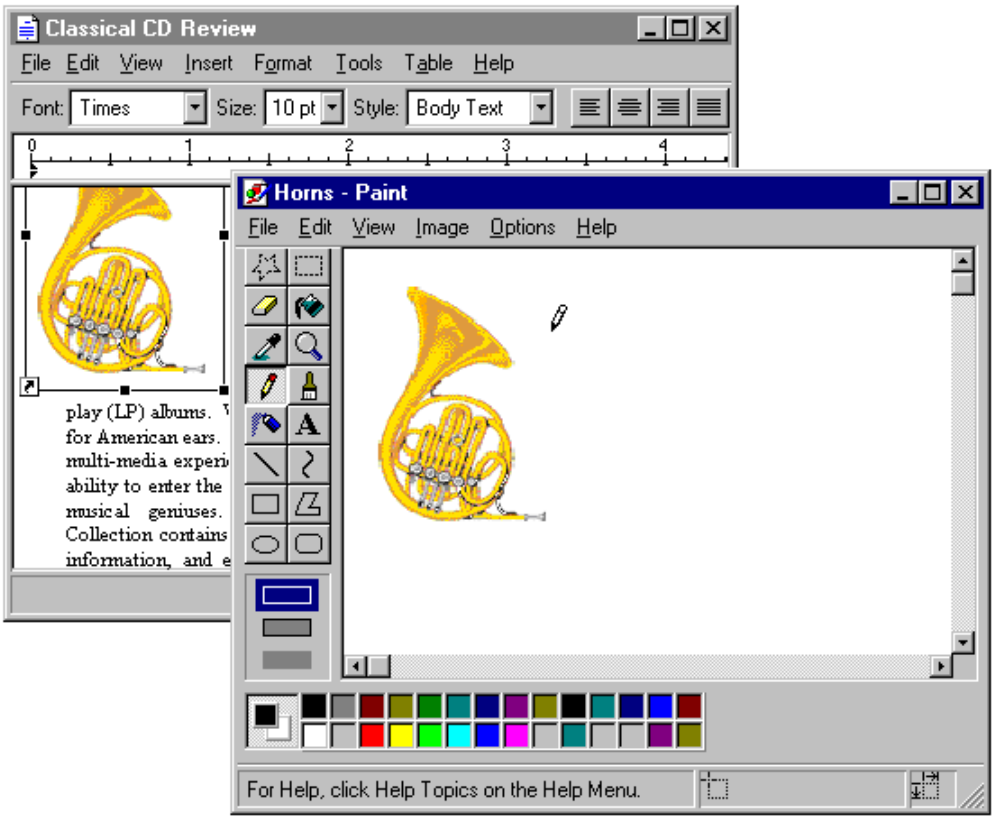


Figure 11.26 Editing a link source

At first glance, editing an OLE linked object seems to appear similar to an opened OLE embedded object. A separate primary window opens displaying the data, but the container of an OLE linked object does not render the link representation using the open hatched pattern because the link source does not reside at this location. The OLE linked object is not the real object, only a stand-in that enables the source to be visually present in other locations. Editing the linked object is functionally identical to opening the link source. Similarly, the title bar text of the link source's window does not use the convention as an open OLE embedded object because the link source is an independent object. Therefore, the windows operate and close independently of each other. If the link source's window is already visible, the OLE linked object notifies the link source to activate, bringing the existing window to the top of the Z order.

Note that the container of the OLE linked object does display messages related to opening the link source. For example, the container displays a message if the link source cannot be accessed.

Automatic and Manual Updating

When the user creates an OLE link, by default it is an automatic link; that is, whenever the source data changes, the link’s visual representation changes without requiring any additional information from the user. Therefore, do not display an “Update Automatic Links Now?” message box. If the update takes a significant time to complete, you can display a message box indicating the progress of the update.

If users wish to exercise control over when links are updated, they can set the linked object’s update property to manual. Doing so requires that the user choose an explicit command to update the link representation. The link can also be updated as a part of the link container's “update fields” or “recalc” action or command.

For more information about updating links automatically or manually, see the section, "Maintaining Links," later in this chapter.

Operations and Links

The operations available for an OLE linked object are supplied by its container and its source. When the user chooses a command supplied by its container, the container application handles the operation. When the user chooses a command supplied by its source, the operation is conceptually passed back to the linked source object for processing. In this sense, activating an OLE linked object activates its source object.

In certain cases, the linked object exhibits the result of an operation; in other cases, the linked source object can be brought to the top of the Z order to handle the operation. For example, executing commands such as Play or Rewind on a link to a sound recording appear to operate on the linked object in place. However, if the user chooses a command to alter the link's representation of its source's content (such as Edit or Open), the link source is exposed and responds to the operation instead of the linked object itself.

A link may play a sound in place, but cannot support editing in place. For a link source to properly respond to editing operations, fully activate the source object (with all of its containing objects and its container). For example, when the user double-clicks a linked object whose default operation is Edit, the source (or its container) opens, displaying the linked source object ready for editing. If the source is already open, the window displaying the source becomes active. This follows the standard convention for activating a window already open; that is, the window comes to the top of the Z order. You can adjust the view in the window, scrolling or changing focus within the window, as necessary, to present the source object for easy user interaction. The linked source window and linked object window operate and close independently of each other.

Note If a link source is contained within a read-only document, edits cannot be saved to the source file.

Types and Links

An OLE linked object includes a cached copy of its source's type at the time of the last update. When the type of a linked source object changes, all links derived from that source object contain the old type and operations until either an update occurs or the linked source is activated. Because out-of-date links can potentially display obsolete operations to the user, a mismatch can occur. When the user chooses a command for an OLE linked object, the linked object compares the cached type with the current type of the linked source. If they are the same, the OLE linked object forwards the operation on to the source. If they are different, the linked object informs its container. In response, the container can either:

- Carry out the new type's operation, if the operation issued from the old link is syntactically identical to one of the operations registered for the source's new type.
- Display a message box, if the issued operation is no longer supported by the link source's new type.

In either case, the OLE linked object adopts the source's new type, and subsequently the container displays the new type's operations in the OLE linked object's menu.

Link Management

An OLE linked object has three properties: the name of its source, its source's type, and the link's updating basis, which is either automatic or manual. An OLE linked object also has a set of commands related to these properties. It is the responsibility of the container of the linked object to provide the user access to these commands and properties. To support this, an OLE container provides a property sheet for all of its OLE objects. You can also include a Links dialog box for viewing and altering the properties of several links simultaneously.

Accessing Properties of OLE objects

Like other types of objects, OLE embedded and linked objects have properties. The container of an OLE object is responsible for providing the interface for access to the object's properties. The following sections describe how to provide user access to the properties of OLE objects.

The Properties Command

Design OLE containers to include a Properties command and property sheets for any OLE objects it contains. If the container application already includes a Properties command for its own native data, you can also use it to support selected OLE embedded or linked objects. Otherwise, add the command to the drop-down and pop-up menu you provide for accessing the other commands for the object, preceded by a menu, as shown in Figure 11.27.

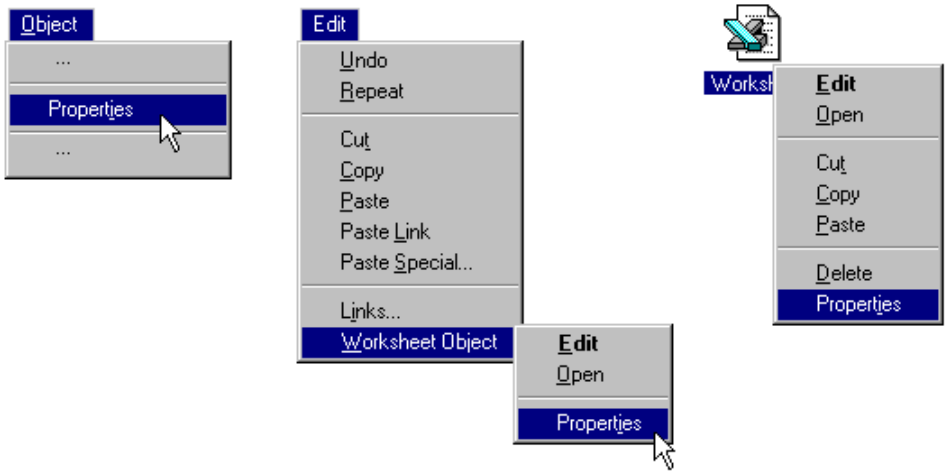


Figure 11.27 The Properties command

When the user chooses the Properties command, the container displays a property sheet containing all the salient properties and values, organized by category, for the selected object. Figure 11.28 shows examples of property sheets for an OLE embedded and linked worksheet object.

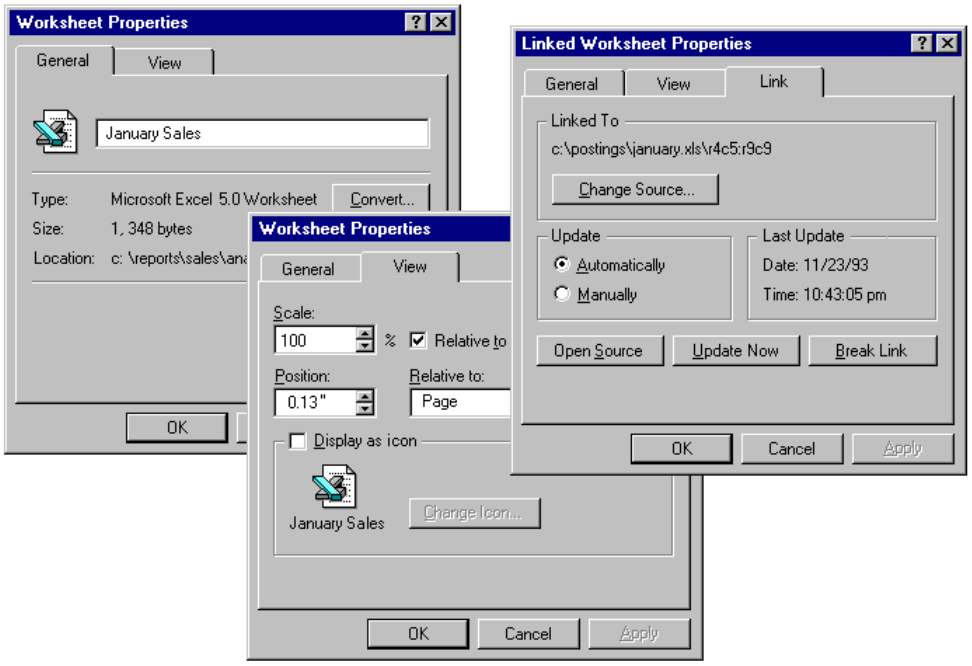


Figure 11.28 OLE embedded and linked object property sheets

Follow the format the system uses for property sheets and the conventions outlined in this guide. Use the short type name in the title bar; for an OLE linked object, precede the name with the word "Linked," as in "Linked Worksheet." Include a General property page displaying the icon, name, type, size, and location of the object. Also include a Convert command button to provide access to the type conversion dialog box. On a View page, display properties associated with the view and presentation of the OLE object within the container. These include scaling or position properties as well as whether to display the object in its content presentation or as an icon. This field includes a Change Icon command button that allows the user to customize the icon presentation of the object.

For OLE linked objects, also include a Link page in its property sheet containing the essential link parameters. For the typical OLE link, include the source name, the Update setting (automatic or manual), the Last Update timestamp, and command buttons that provide the following link operations:

- Break Link effectively disconnects the selected link.
- Update Now forces the selected link to connect to its sources and retrieve the latest information.
- Open Source opens the link source for the selected link.
- Change Source invokes a dialog box similar to the common Open dialog box to allow the user to respecify the link source.

The Links Command

OLE containers can include a Links command that provides access to a dialog box for displaying and managing multiple links. Figure 11.29 shows the Links dialog box. The list box in the dialog box displays the links in the container. Each line in the list contains the link source's name, the link source's object type (short type name), and whether the link updates automatically or manually. If a link source cannot be found, “Unavailable” appears in the update status column.

Note The Microsoft Win32 Software Development Kit includes the Links dialog box and other OLE-related dialog boxes described in this chapter.

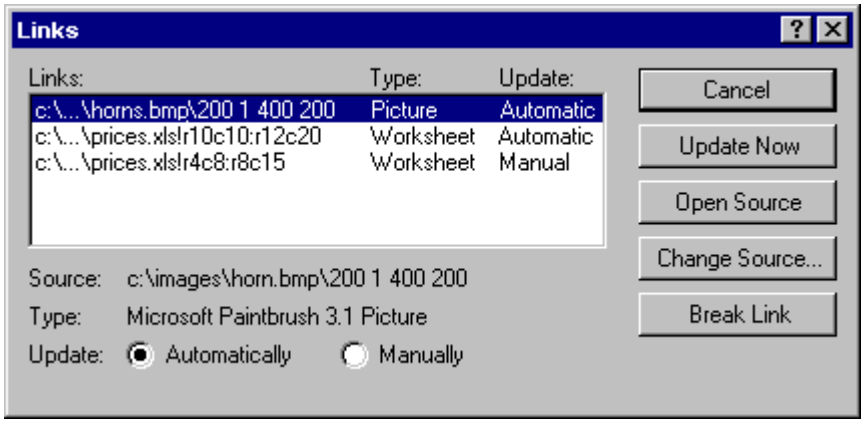


Figure 11.29 The Links dialog box

If the user chooses the Links command when the current selection includes a linked object (or objects), display that link (or links) as selected in the Links dialog box and scroll the list to display the first selected link at the top of the list box.

Allow 15 characters for the short type name field, and enough space for Automatic and Manual to appear completely. As the user selects each link in the list, its type, name, and updating basis appear in their entirety at the bottom of the dialog box. The dialog box also includes link management command buttons included in the Link page of OLE linked object property sheets: Break Link, Update Now, Open Source, and Change Source.

Define the Open Source button to be the default command button when the input focus is within the list of links. Support double-clicking an item in the list as a shortcut for opening that link source.

The Change Source button enables the user to change the source of a link by selecting a file or typing a filename. If the user enters a source name that does not exist and chooses OK, a message box is displayed with the following message, as shown in Figure 11.30.

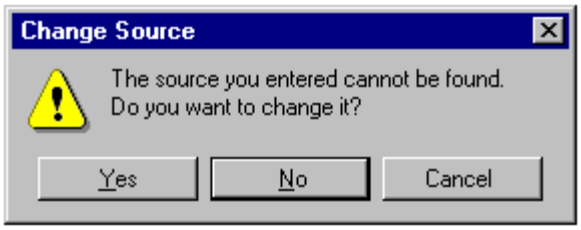


Figure 11.30 A message box for an invalid source

If the user chooses Yes, display the Change Source dialog box to correct the string. If the user chooses No, store the unparsed display name of the link source until the user links successfully to a newly created object that satisfies the dangling reference. The container application can also choose to allow the user to connect only to valid links. If the user chooses Cancel, remove the message box and return input focus to the Links dialog box.

If the user changes a link source or its directory, and other linked objects in the same container are connected to the same original link source, the container may offer the user the option to make the changes for the other references. To support this option, use the message box, as shown in Figure 11.31.

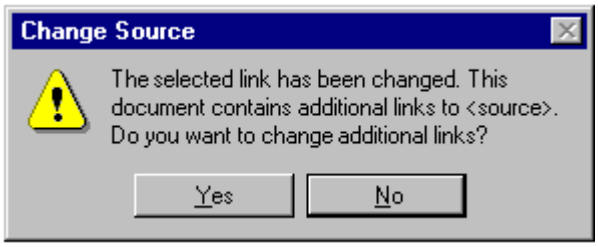


Figure 11.31 Changing additional links with the same source

Converting Types

Users may want to convert an object's type, so they edit the object with a different application. To support the user's converting an OLE object from its current type to another registered type, provide a Convert dialog box, as shown in Figure 11.32. The user accesses the Convert dialog box by including a Convert button beside the Type field in an object's property sheet.

Note Previous guidelines recommended including a Convert command on the menu for a selected OLE object. You may continue to support this; however, providing access through the property sheet of the object is the preferred method.

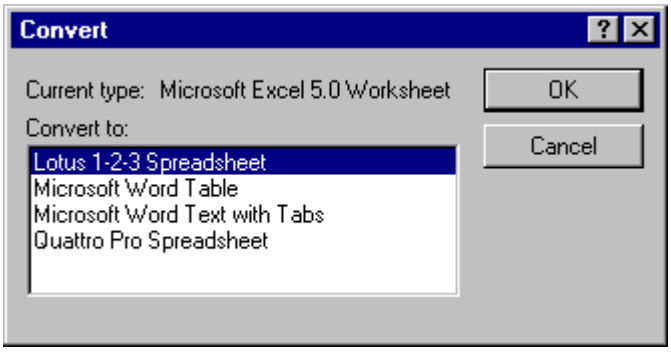


Figure 11.32 The Convert dialog box

This dialog box displays the current type of the object and a list box with all possible conversions. This list is composed of all types registered as capable of reading the selected object's format, but this does not necessarily guarantee the possibility of reverse conversion. The list should not contain the object's type, because that would result in null operation. If the user selects a new type from the list and chooses the OK button, the selected object is converted immediately to the new type. If the object is open, the container closes it before beginning the conversion.

Make sure the application that supplies the conversion does so with minimal impact in the user interface. That is, avoid displaying your application's primary window, but do provide a progress indicator message box with appropriate controls so that the user can monitor or interrupt the conversion process.

For more information about progress message boxes, see Chapter 8, "Secondary Windows."

If the conversion of the type could result in any lost data or information, the application you use to support the type conversion should display a warning message box indicating that data will be lost and request confirmation by the user before continuing. Make the message as specific as possible about the nature of the information that might be lost; for example, "Text properties will not be preserved." If the conversion will result in no data loss, the warning message is not necessary.

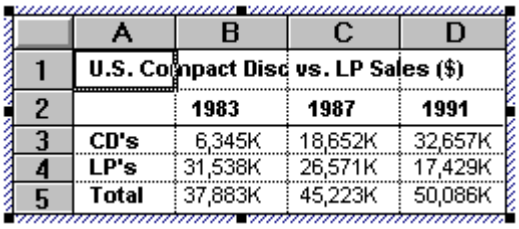
The Convert button on an object's property sheet should be disabled for linked objects, because conversion must occur on the link source. In any case, disable the Convert button for all objects that do not have any conversion options; that is, no other type is capable of converting the selected object.

Using Handles

A container displays handles for an OLE embedded or linked object when the object is selected individually. When an object is selected and not active, only the scaling of the object (its cached metafile) can be supported. If a container uses handles for indicating selection but does not support scaling of the image, use the hollow form of handles.

For more information about the appearance of handles, see Chapter 13, "Visual Design."

When an OLE embedded object is activated for OLE visual editing, it displays its own handles. The active object also determines which operation to support when the user drags a handle. Display the handles within the active hatched pattern, as shown in Figure 11.33.



	A	B	C	D
1	U.S. Compact Disc vs. LP Sales ($)			
2		1983	1987	1991
3	CD's	6,345K	18,652K	32,657K
4	LP's	31,538K	26,571K	17,429K
5	Total	37,883K	45,223K	50,086K

Figure 11.33 An active OLE embedded object with handles

The interpretation of dragging the handle is defined by the OLE embedded object's application. The recommended operation is cropping, where you expose more or less of the OLE embedded object's content and adjust the viewport. If cropping is inappropriate or unsupportable, use an operation that better fits the context of the object or simply support scaling of the object. If no operation is meaningful, but handles are required to indicate selection while activated, use the hollow handle appearance.

Undo Operations for Active and Open Objects

Different objects (that is, different underlying applications) take control of a window during OLE visual editing, so managing commands like Undo or Redo present a question: How are the actions performed within an edited OLE embedded object reconciled with actions performed on the native data of the container with the Undo command? The recommended undo model is a single undo stack per open window—that is, all actions that can be reversed, whether generated by OLE embedded objects or their container, accumulate on the same undo state sequence. Therefore, choosing Undo from either the container's menus or an active object's menus reverses the last undoable action performed in that open window, regardless of whether it occurred inside or outside the OLE embedded object. If the container has the focus and the last action in the window occurred within an OLE embedded object, when the user chooses Undo, activate the embedded object, reverse the action, and leave the embedded object active.

The same rule applies to open objects—that is, objects that have been opened into their own window. Because each open window manages a single stack of undoable states, actions performed in an open object are local to that object’s window and consequently must be undone from there; actions performed in the open object (even if they create updates in the container) do not contribute to the undo state of the container.

Carrying out a registered command of a selected, but inactive, object (or using a shortcut equivalent) is not a reversible action; therefore, it does not add to a container’s undo stack. This includes opening an object into another window for editing. For example, if the user opens an object, this action cannot be undone from its container. The resulting window must be closed directly to remove it.

Figure 11.34 shows two windows: Container Window A, which has an active OLE embedded object, and an open embedded object in Window B. Between the two windows, nine actions have been performed in the order and at the location indicated by the numbers. The resulting undo stacks are displayed beneath the windows.

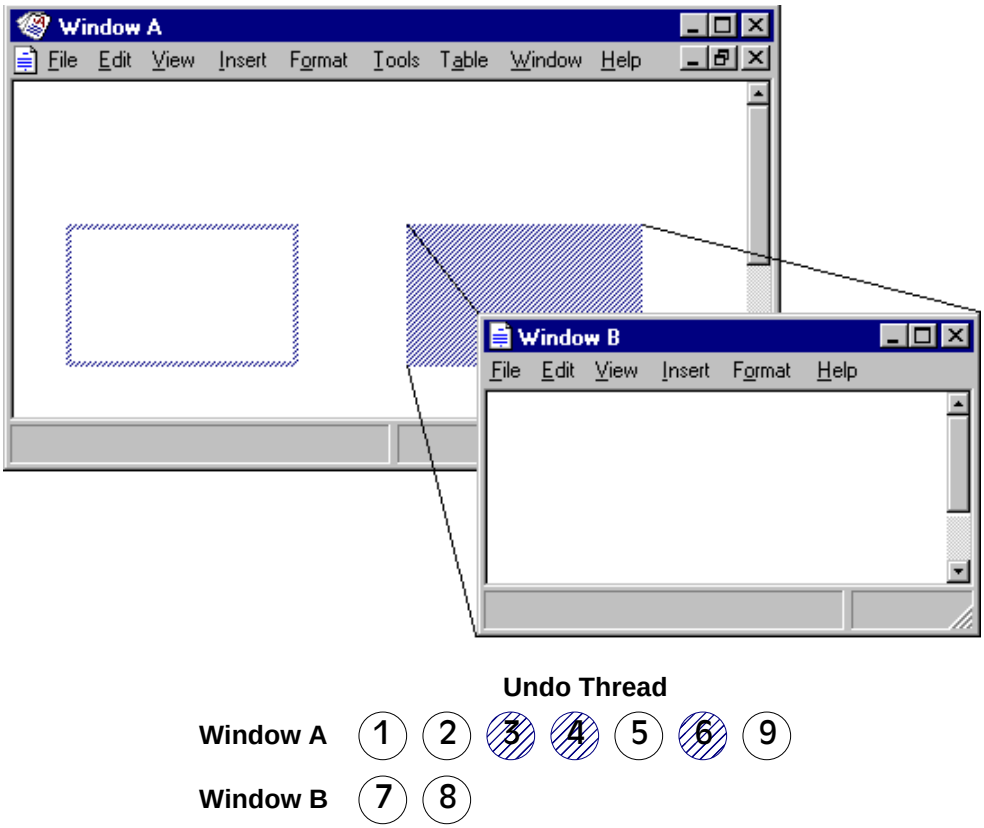


Figure 11.34 Undo stacks forive and open OLE embedded objects

The sequence of undo states shown in Figure 11.34 does not necessarily imply an *n*-level undo. It is merely a timeline of actions that can be undone at 0, 1, or more levels, depending on what the container-object cooperation supports.

The active object actions and native data actions within Window A have been serialized into the same stack, while the actions in Window B have accumulated onto its own separate stack.

The actions discussed so far apply to a single window, not to actions that span multiple windows, such as OLE drag and drop. For a single action that spans multiple windows, the ideal design allows the user to undo the action from the last window involved. This is because, in most cases, the user focuses on that window when the mistake is recognized. So if the user drags and drops an item from Window A into Window B, the action appends to Window B's undo thread, and undoing it undoes the entire OLE drag and drop operation. Unfortunately, the system does not support multiple window undo coordination. So for a multiple window action, create independent undo actions in each window involved in the action.

Displaying Messages

This section includes recommendations about other messages to display for OLE interaction using message boxes and status line messages. Use the following messages in addition to those described earlier in this chapter.

Object Application Messages

Display the following messages to notify the user about situations where an OLE object's application is not accessible.

Object's Application Cannot Run Standalone

Some OLE objects are designed to be used only as components within a container and have no value in being opened directly. If the user attempts to open or run an OLE object's application that cannot run as a standalone application, display the message box shown in Figure 11.35.

Figure 11.35 Object's application cannot be run standalone message

Object's Application Location Unknown

When the user selects an entry from the Insert Object dialog box or activates an object, and the container cannot locate the requested object's application, display the message shown in Figure 11.36.

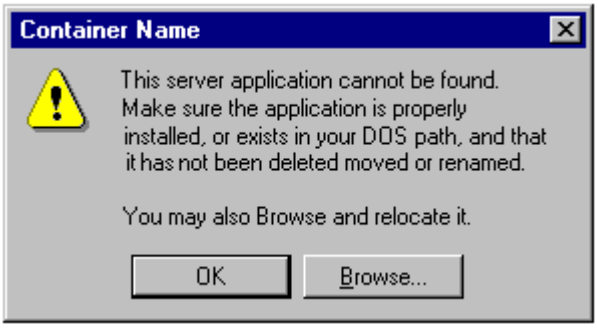


Figure 11.36 Object's application location is unknown message

When the user chooses the Browse button, display the common Open dialog box. Enter the user-supplied path in the registry as the new object application path and filename.

If a container supports inside-out activation for an object, display this message when the user tries to interact with that object, not when the container is opened. This avoids the display of the message to the user who only intends to view the content.

Object's Application Unavailable

An object's application can be unavailable for several reasons. For example, it can be busy printing, waiting for user input to a modal message box, or the application has stopped responding to the system. If the object's application is not available, display the message box shown in Figure 11.37.

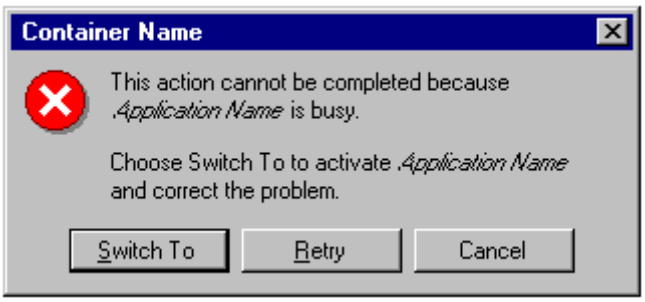


Figure 11.37 Object's application is unavailable message

Object's Application Path Unavailable

If the path for the object's application is invalid because a network is unavailable, display the message box shown in Figure 11.38.

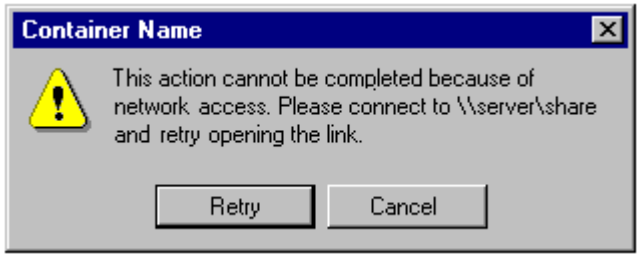


Figure 11.38 Object's application path is unavailable message

Object's Type Unregistered

If the user attempts to activate an object whose type is not registered in the registry, display the message box shown in Figure 11.39.

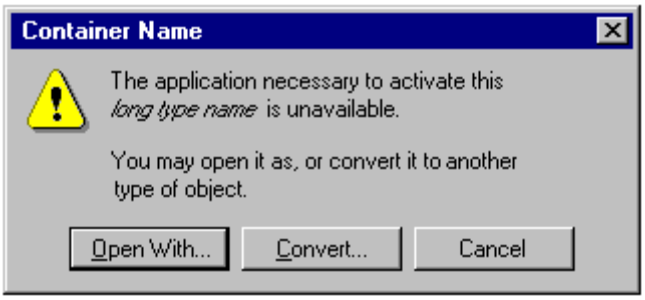


Figure 11.39 Object type is not registered message

Choosing the Convert button displays the Convert dialog box. Choosing the Open With button displays a dialog box with a list of current types the user can use to edit the object. Ideally, an application that registers the type should be able to read and write that format without any loss of information. If it cannot preserve the information of the original type, the application handling the type emulation displays a message box warning the user about what information it cannot preserve and optionally allows the user to convert the object's type.

OLE Linked Object Messages

Display the following messages to notify the user about situations related to interaction with OLE linked objects.

Link Source Files Unavailable

When a container requests an update for its OLE linked objects, either because the user chooses an explicit Update command or as the result of another action such as a Recalc operation, if the link source files for some OLE links are unavailable to provide the update, display the message box shown in Figure 11.40.

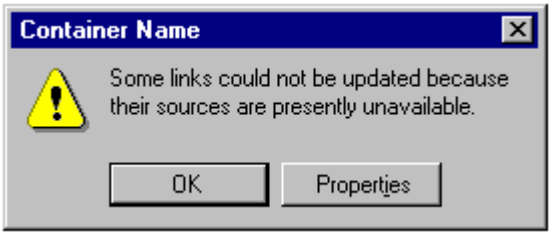


Figure 11.40 Link source files are unavailable message

Include two buttons, OK and Properties. When the user chooses the OK button, close the dialog box without updating the links. Choosing the Properties button displays a property sheet for the link (see Figure 11.28) with "Unavailable" in the Update field. The user can then use the Change Source button to search for the file or choose other commands related to the link.

Optionally, you can also include a Links button in the message box. When the user chooses this button, display your Links dialog box, following the same conventions as for the property sheet.

Similarly, if the user issues a command to an OLE linked object with an unavailable source, display the warning message shown in Figure 11.41. Display the OLE linked object's update status as "Unavailable."

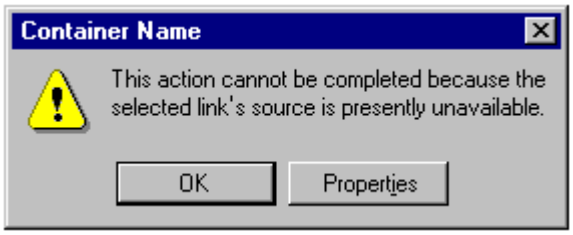


Figure 11.41 Selected link source is unavailable message

Link Source Type Changed

If a link source's type has changed, but it is not yet reflected for an OLE linked object, and the user chooses a command that does not support the new type, display the message box shown in Figure 11.42.

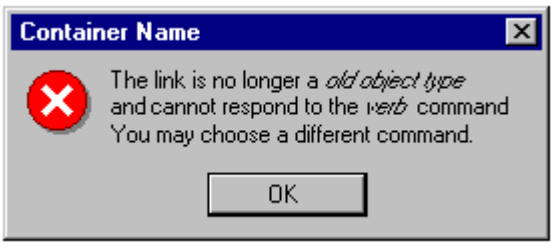


Figure 11.42 Link source's type has changed message

Link Updating

While links are updating, display the progress indicator message box shown in Figure 11.43. The Stop button interrupts the update process and prevents any updating of additional links.

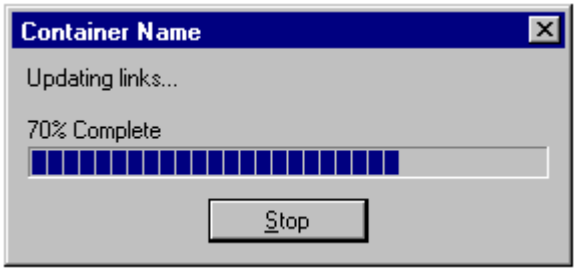


Figure 11.43 Progress indicator while links update message

Status Line Messages

Table 11.3 lists suggested status line messages for commands on the primary container menu (commonly the File menu) of an opened object.

Table 11.3 Primary Container Menu Status Line Messages

Command	Status line message
Update *container-document*	Updates the appearance of this *full type name* in *container-document.*
Close & Return To *container-document*	Closes *object name* and returns to *container-document.*
Save Copy As	Saves a copy of *descriptive type name* in a separate file.
Exit & Return To *container-document*	Exits *object application* and returns to *container-document.*

Note If the open object is within an MDI application with other open documents, the Exit & Return To command should simply be "Exit". There is no guarantee of a successful Return To *container-document* after exiting, because the container might be one of the other documents in that MDI instance.

Table 11.4 lists the recommended status line messages for the Edit menu of containers of OLE embedded and linked objects.

Table 11. 4 Edit Menu Status Line Messages

Command	Status line message
Paste *Object Name*[1]	Inserts the content of the Clipboard as O*bject Name*.
Paste Special	Inserts the content of the Clipboard with format options.
Paste Link [To] *Object Name*[1]	Inserts a link to *Object Name*.
Paste Shortcut [To] O*bject Name*[1]	Inserts a shortcut icon to *Object Name*.
Insert Object	Inserts a new object.
[Linked]*Object Name*[1] [Object]➔	Applies the following commands to *object name*.
[Linked] O*bject Name*[1] [Object]➔ *Command*	Varies based on command.
[Linked] *Object Name*[1] [Object]➔ Properties	Allows properties of *object name* to be viewed or modified.
Links	Allows links to be viewed, updated, opened, or removed.

[1]*object name* may be either the object's short type name or its filename.

Table 11.5 lists other related status messages.

Table 11. 5 Other Status Line Messages

Command	Status line message
Show Objects	Displays the borders around objects (toggle).
Select object (when the user selects an object)	Double-click or press ENTER to *default - command object name* or *full type name*.

Note The default command stored in the registry contains an ampersand character (&) and the access key indicator; these must be stripped out before the verb is displayed on the status line.

CHAPTER 12

User Assistance

User assistance is an important part of a product's design. A well-designed Help interface provides a user with assistance upon demand, but the assistance must be simple, efficient, and relevant so that a user can obtain it without becoming lost in the Help interface. A user wants to accomplish a task—a Help interface design should assist in that objective without being intrusive. This chapter provides a description of the system support to create your own user assistance support and guidelines for implementation.

Contextual User Assistance

A contextual form of user assistance provides information about a particular object and its context. It answers questions such as “What is this?" and "Why would I use it?” This section covers some of the basic ways to support contextual user assistance in your application.

Context-Sensitive Help

The What’s This? command, as shown in Figure 12.1, supports a user obtaining contextual information about any object on the screen, including controls in property sheets and dialog boxes. This form of contextual user assistance is referred to as *context-sensitive Help*. You can support user access to this command by including:

- A What’s This? command from the Help drop-down menu.
- A What’s This? button on a toolbar.
- A What’s This? button on the title bar of a secondary window.
- A What’s This? command on the pop-up menu for the specific object.

Figure 12.1 Different methods of accessing What's This?

Design your application so that when the user chooses the What’s This? command from the Help drop-down menu or clicks a What’s This? button, the system is set to a temporary mode. Change the pointer’s shape to reflect this mode change, as shown in Figure 12.2. The SHIFT+F1 combination is the shortcut key for this mode.

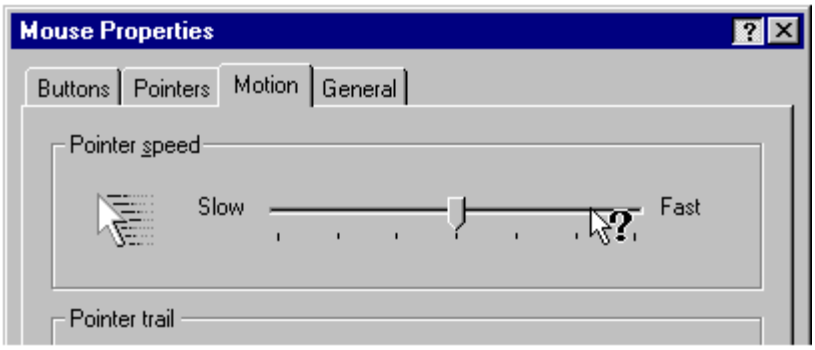


Figure 12.2 A context-sensitive Help pointer

Display the context-sensitive Help pointer only over the window that provides context-sensitive Help; that is, only over the active window from which the What's This? command was chosen.

In this mode, when the user clicks an object with mouse button 1 (for pens, tapping), display a context-sensitive Help pop-up window for that object. The context-sensitive Help window provides a brief explanation about the object and how to use it, as shown in Figure 12.3. Once the context-sensitive Help window is displayed, return the pointer and pointer operation to its usual state.

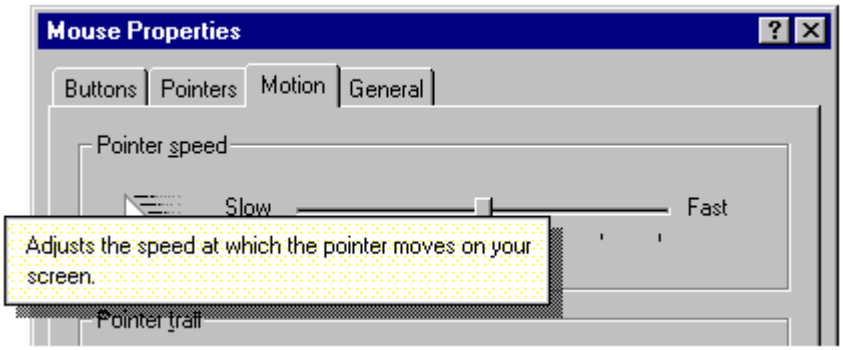


Figure 12.3 A pop-up window for context-sensitive Help

If the user presses a shortcut key that applies to a window that is in contextual Help mode, you can display a contextual Help pop-up window for the command associated with that shortcut key.

However, there are some exceptions to this interaction. First, if the user chooses a menu title, either in the menu bar or a cascading menu, maintain the mode and do not display the context-sensitive Help window until the user chooses a menu item. Second, if the user clicks the item with mouse button 2 and the object supports a pop-up menu, maintain the mode until the user chooses a menu item or cancels the menu. If the object does not support a pop-up menu, the interaction should be the same as clicking it with mouse button 1. Finally, if the chosen object or location does not support context-sensitive Help or is otherwise an inappropriate target for context-

sensitive Help, continue to display the context-sensitive Help pointer and maintain the context-sensitive Help mode.

If the user chooses the What's This? command a second time, clicks outside the window, or presses the ESC key, cancel the context-sensitive Help mode. Restore the pointer to its usual image and operation in that context.

When the user chooses the What's This? command from a pop-up menu (as shown in Figure 12.4), the interaction is slightly different. Because the user has identified the object by clicking mouse button 2, there is no need for entering the context-sensitive Help mode. Instead, immediately display the context-sensitive Help pop-up window for that object.

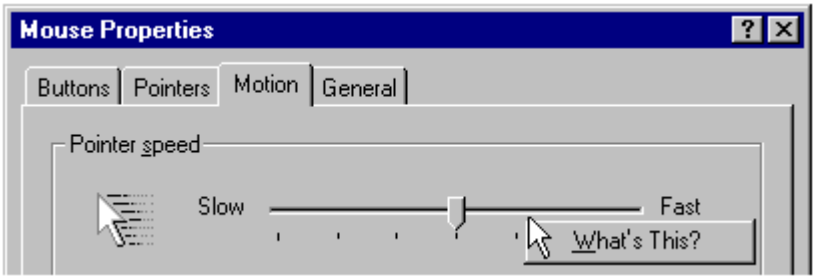


Figure 12.4 A pop-up menu for a control

The F1 key is the shortcut key for this form of interaction; that is, pressing F1 displays a context-sensitive Help window for the object that has the input focus.

Guidelines for Writing Context-Sensitive Help

When authoring context-sensitive Help information, you are answering the question "What is this?" Start each topic with a verb — for example, "Adjusts the speed of your mouse," "Click this button to close the window," or "Type in a name for your document." When describing a function or object, use words that explain the function or object in common terms. For example, instead of "Undoes the last action," say "Reverses the last action."

In the explanation, you might want to include "why" information. You can also include "how to" information, but this information is better handled by providing access to task-oriented Help. Keep your information brief, but as complete as possible so that the Help window is easy and quick to read.

You can provide context-sensitive Help information for your supported file types by registering a What's This? command for the type, as shown in Figure 12.5. This allows the user to choose the "What's This?" command from the file icon's pop-up menu to get information about an icon representing that type. When defining this Help information, include the long type name and a brief description of its function, using the previously described guidelines.

For more information about registering commands for file types and about type names, see Chapter 10, "Integrating with the System."

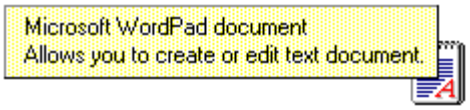


Figure 12.5 Help information for an icon

Tooltips

Another form of contextual user assistance are tooltips. *Tooltips* are small pop-up windows that display the name of a control when the control has no text label. The most common use of tooltips is as toolbar buttons that have graphic labels, as shown in Figure 12.6.

Figure 12.6 A tooltip for a toolbar button

Display a tooltip after the pointer, or pointing device, remains over the button for a short period of time. Base the time-out on the system timing metric (XXX). The tooltip remains displayed until the user presses the button or moves off of the control, or after another time-out. If the user moves the pointer directly to another control supporting a tooltip, ignore the time-out and display the new tooltip immediately, replacing the former one.

For more information about the appearance of toolips, see Chapter 13, “Visual Design.”

Status Bar Messages

You can also use a status bar to provide contextual user assistance. However, because it is important to support the user's choice of displaying a status bar, avoid using it for displaying information or access to functions that are essential to basic operation and not provided elsewhere in the application's interface.

In addition to displaying state information about the context of the activity in the window, you can display descriptive messages about menu and toolbar buttons, as shown in Figure 12.7. Like tooltips, the window typically must be active to support these messages. When the user moves the pointer over a toolbar button or presses the mouse button on a menu or button, display a short message the describing use of the associated command.
For more information about the status bar control, see Chapter 7, “Menus, Controls, and Toolbars.”

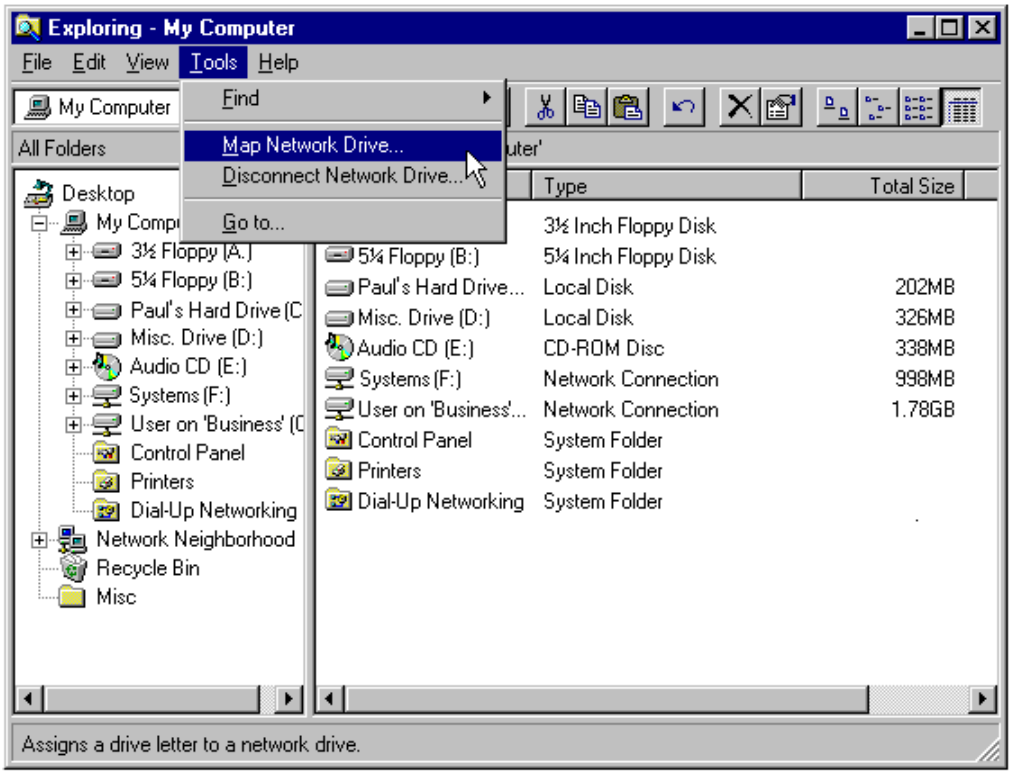


Figure 12.7 A status bar message for a menu command

A status bar message can include a progress indicator control or other forms of feedback about an ongoing process, such as printing or saving a file, that the user initiated in the window. While you can display progress information in a message box, you may want to use the status bar for background processes so that the window's interface is not obscured by the message box.

Guidelines for Writing Status Bar Messages

When writing status bar messages, begin the text with a verb in the present tense and use familiar terms, avoiding jargon. For example, say “Cuts the selection and puts it on the Clipboard.” Try to be as brief as possible so the text can be easily read, but avoid truncation.

Be constructive, not just descriptive, informing the user about the purpose of the command. When describing a command with a specific function, use words specific to the command. If the scope of the command has multiple functions, try to summarize. For example, say “Contains commands for editing and formatting your document.”

When defining messages for your menu and toolbar buttons, don't forget their unavailable, or disabled, state. Provide an appropriate message to explain why the item is not currently available. For example, say "This command is not available because no text is selected."

The Help Command Button

You can also provide contextual Help for a property sheet, dialog box, or message box by including a Help button in that window, as shown in Figure 12.8.

Figure 12.8 A Help button in a secondary window

This differs from the “What’s This?” form of Help because it provides an overview, summary assistance, or explanatory information for that window. For example, the Help button on a property sheet provides information about the properties included in the property sheet; for a message box, it provides more information about causes and remedies for the reason the message was displayed.

When the user presses the Help command button, display the Help information in a Help secondary window, rather than a context-sensitive Help pop-up window, as shown in Figure 12.9.

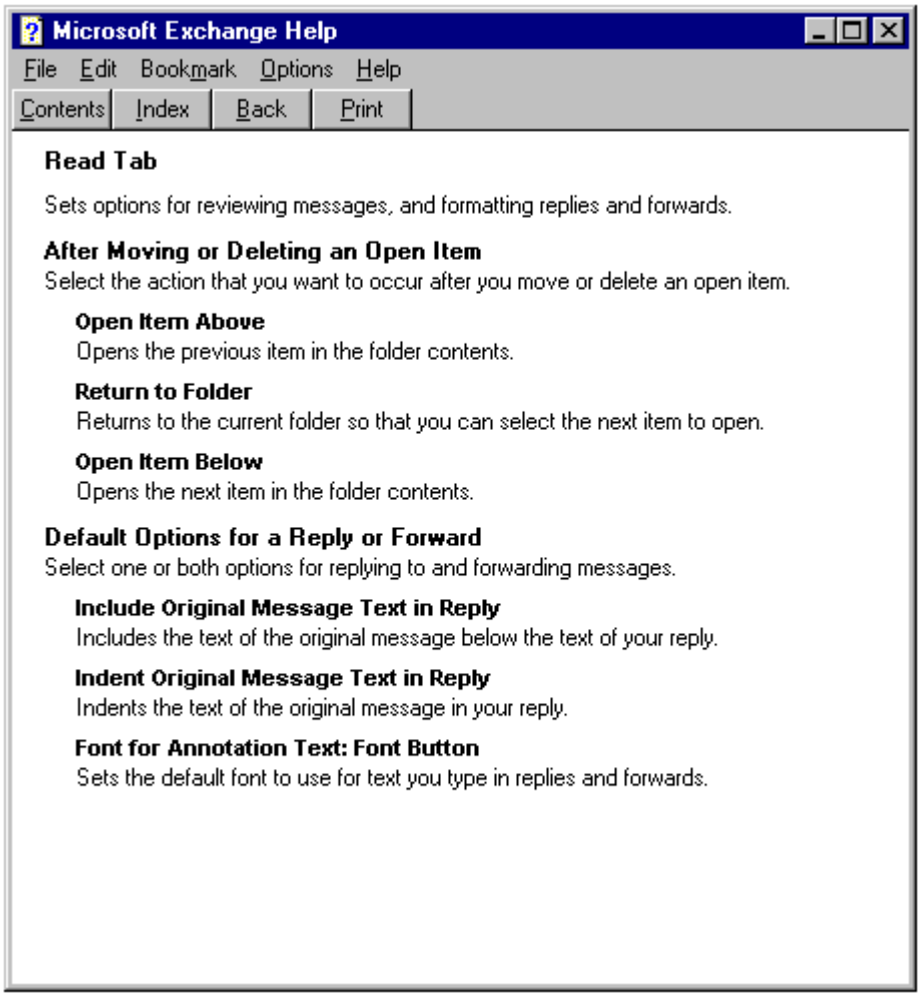


Figure 12.9 A Help secondary window

Task Help

Task-oriented help provides the steps for carrying out a task. It can involve a number of procedures. You present task-oriented Help in task topic windows.

Task Topic Windows

Help task topic windows are displayed as primary windows. The user can size this window like any other primary window.

Note The window style is referred to as a primary window because of its appearance and operation. In technical documentation, this window style is sometimes referred to as a Help secondary window.

Task topic windows include a set of command buttons at the top of the window (as shown in Figure 12.10) that provide the user access to the tabbed pages of the Help Topics browser, the previously selected topic, and other Help options, such as copying and printing a topic. You can define which buttons appear by defining them in your Help files.

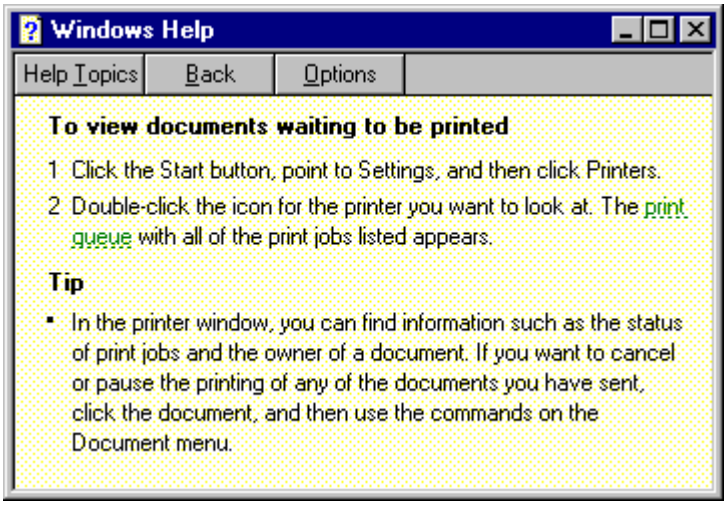


Figure 12.10 A window for a Help task topic

While you can define the size and location of a Help task topic window to the specific requirements of your application, it is best to size and position the window so as to cover the minimum of space, but make it large enough to allow the user to read the topic, preferably without having to scroll the window.

Like tooltips, the interior color of a task topic window uses the system color setting for Help windows. This allows the user to more easily distinguish the Help topic from their other windows.

For more information about the visual design of Help windows, see Chapter 13, "Visual Design."

Guidelines for Writing Task Help Topics

The buttons that appear in a task topic window are defined by your Help file. At a minimum, you should provide a button that displays the Help Topics browser dialog box, a Back button to return the user to the previous topic, and buttons that provide access to other functions, such as Copy and Print. You can provide access to the Help Topics browser dialog box by including a Help Topics button. This displays the Help Topics browser window on the tabbed page that the user was viewing when the window was last displayed. While this is the most common form of access to the Help Topics browser window, alternatively you can include buttons, such as Contents and Index, that correspond to the tabbed pages to provide the user with direct access to those pages when the dialog box is displayed.

As with context-sensitive Help, when writing Help task information topics, make them complete, but brief. Focus on “how” information rather than “what” or “why.” If there are multiple alternatives, pick one method — usually the simplest, most common method for a specific procedure. If you want to include information on alternative methods, provide access to them through other choices or commands.

If you keep the procedure to four or fewer steps, the user will not need to scroll the window. Avoid introductory, conceptual, or reference material in the procedure.

Also, take advantage of the context of a procedure. For example, if a property sheet includes a slider control that is labeled “Slow” at one end and “Fast” at the other, be contextually concise. Say “Move the slider to adjust the speed” instead of “To increase the speed, move the slider to the right. To decrease the speed, move the slider to the left.” If you refer to a control by its label, capitalize each word in the label, even though the label has only the first word capitalized.

Shortcut Buttons

Help task topic windows can also include a shortcut or “do it” button that provides the user with a shortcut or automated form of performing a particular step, as shown in Figure 12.11. For example, use this to automatically open a particular dialog box, property sheet, or other object so that the user does not have to search for it.

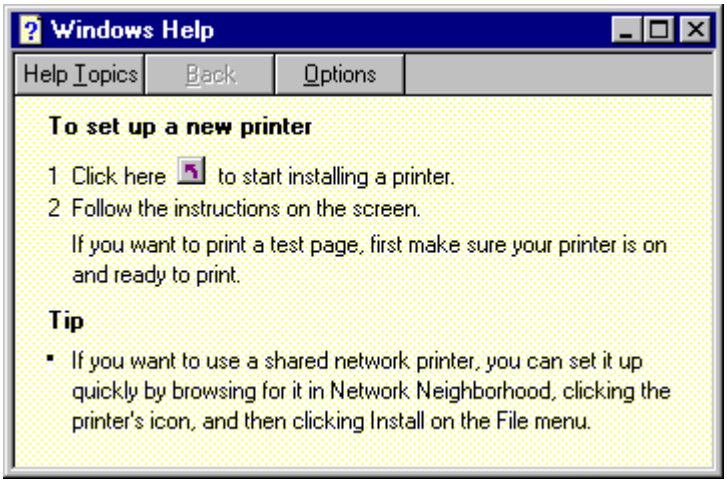


Figure 12.11 A Task Topic with a shortcut button

Reference Help

Reference Help is a form of Help information that serves more as online documentation. Use reference Help to document the features of a product or as a user's guide to a product. Often the use determines the balance of text and graphics used in the Help file. Reference-oriented documentation typically includes more text and follows a consistent presentation of information. User's guide documentation typically organizes information by specific tasks and may include more illustrations.

The Reference Help Window

When designing reference Help, use a Help primary window style (sometimes called a "main" Help window), as shown in Figure 12.12, rather than the context-sensitive Help pop-up windows or task topic windows.

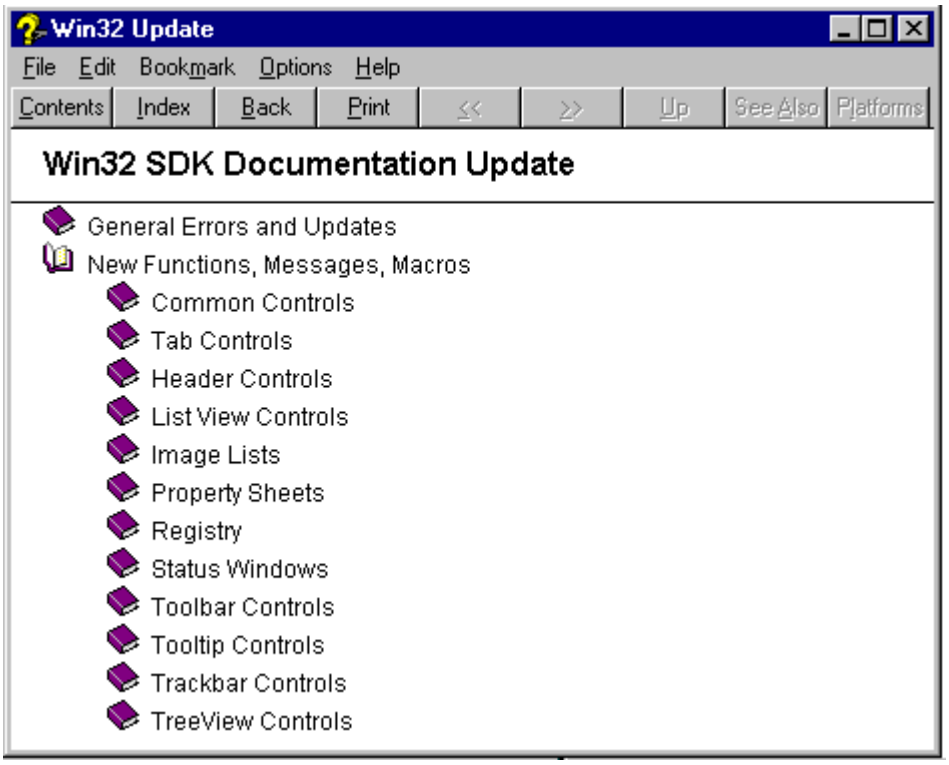


Figure 12.12 A reference Help window

You can provide access to reference Help in a variety of ways. The most common is as an explicit menu item in the Help drop-down menu, but you can also provide access using a toolbar button, or even as a specific file object (icon).

A reference Help window includes a menu bar, with File, Edit, Bookmark, Options, and Help entries and a toolbar with Contents, Index, Back, and Print buttons. (The system provides these features by default for a "main" Help window.) These features support user functions such as opening a specific Help file, copying and printing topics, creating annotations and bookmarks for specific topics, and setting the Help window's properties. You can add other buttons to this window to tailor your online documentation to fit your particular user needs.

While the reference Help style can provide information similar to that provided in contextual Help and task Help, these forms of Help are not exclusive of each other. Often the combination of all these items provides the best solution for user assistance. They can also be supplemented with other forms of user assistance.

Guidelines for Writing Reference Help

Reference Help topics can include text, graphics, animations, video, and audio effects. Follow the guidelines included throughout this guide for recommendations on using these elements in the presentation of information. In addition, the system provides some special support for Help topics.

For more information about authoring Help files, see the help provided with the Windows Help Compiler in the Microsoft Win32 SDK.

Adding Menus and Toolbar Buttons

You can author additional menus and buttons to appear in the reference Help window. However, you cannot remove existing menus.

Because reference Help files typically include related topics, include Previous Topic and Next Topic browse buttons in your Help window toolbar. Another common button you may want to include is a See Also button that either displays a pop-up window or a dialog box with the related topics. Other common buttons include Up for moving to the parent or overview topic and History to display a list of the topics the user has viewed so they can return directly to a particular topic.

Make toolbar buttons contextual to the topic the user is viewing. For example, if the current topic is the last in the browse chain, disable the Next Topic button. When deciding whether to disable or remove a button, follow the guidelines defined in this guide for menus.

For more information about disabling and removing menu entries, see Chapter 7, "Menus, Controls, and Toolbars."

Topic Titles

Always provide a title for the current topic. The title identifies the application and context of the topic and provides the user with a landmark within the Help system.

Nonscrolling Regions

If your topics are very long, you may want to include a nonscrollable region in your Help file. A nonscrolling region allows you to keep the topic title and other information visible when the user scrolls. A nonscrolling region appears with a line at its bottom edge to delineate it from the scrollable area. Display the scroll bar for the scroll area of the topic so that its top appears below the nonscrolling region, not overlapped within that region.

Jumps

A jump is a button or hot spot area that triggers an event when the user clicks on it. You can use a jump as a one-way navigation link from one topic to another, either within the same topic window, to another topic window, or a topic in another Help file.

You can also use jumps to display a pop-up window. As with pop-up windows for context-sensitive Help, use this form of interaction to support a definition or explanatory information about the word or object that the user clicks.

Jumps can also carry out particular commands. Shortcut buttons used in Help task topics are this form of a jump.

You need to provide visual indications to distinguish a jump from noninteractive areas of the window. You can do this by formatting a jump as a button, changing the pointer image to indicate an interactive element, formatting the item with some other visual distinction such as color or font, or a combination of these methods. The default system presentation for text jumps is green underlined text.

The Help Topics Browser

The Help Topics browser dialog box provides user access to Help information. To open this window, include a Help Topics menu item on the Help drop-down menu. Alternatively, you can include menu commands that open the window to a particular tabbed page—for example, Contents, Index, and Find Topic.

In addition, provide a Help Topics button in the toolbar of a Help Topics window. When the user chooses this button, display the Help Topics browser window as the last page the user accessed. If you prefer, provide Contents, Index, and Find Topic buttons for direct access to a specific page.

The Help Topic Tabs

Opening the Help Topics window displays a set of tabbed pages. The default pages include Contents, Index, and Find tabs. You can author additional tabs.

The Contents page displays the list of topics organized by category, as shown in Figure 12.13. A book icon represents a category or group of related topics and a page icon represents an individual topic. You can nest topic levels, but avoid nesting topics too deeply as this can make access cumbersome.

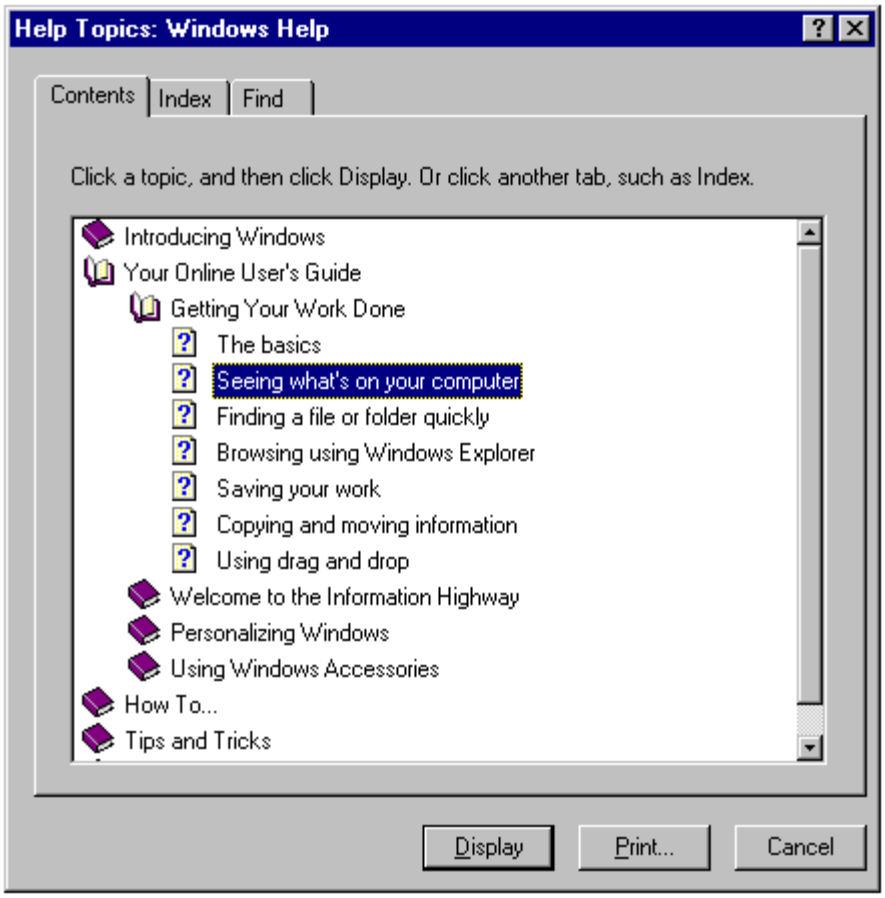


Figure 12.13 The Contents page of the Help topics browser

The buttons at the bottom of the page allow the user to open or close a "book" of topics and display a particular topic. The Print button prints either a “book” of topics or a specific topic depending on which the user selects. The outline also supports direct interaction for opening the outline or a topic.

The Index page of the browser organizes the topics by keywords that you define for your topics, as shown in Figure 12.14.

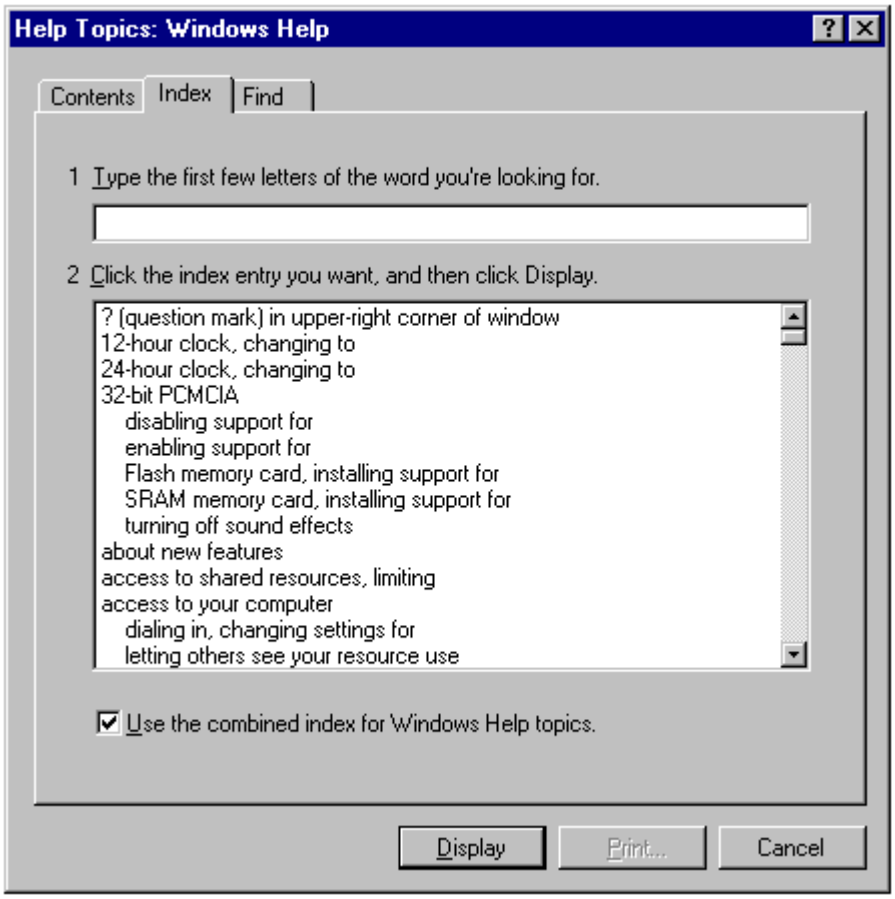


Figure 12.14 The Index age of the Help Topics browser

The user can enter a keyword or select one from the list. Choosing the default button displays the topic associated with that keyword. If there are multiple topics that use the same keyword, then another secondary window is displayed that allows the user to choose from that set of topics, as shown in Figure 12.15.

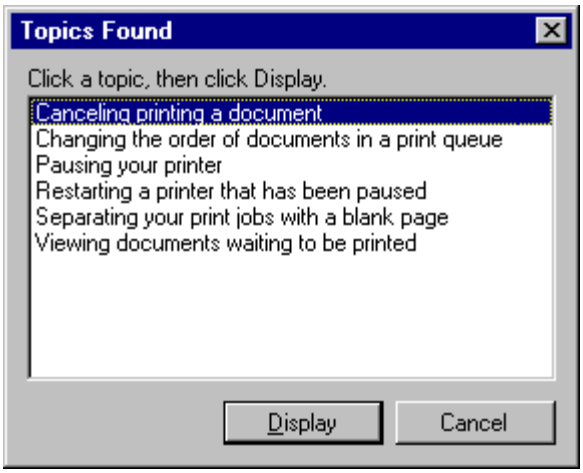


Figure 12.15 The Help topics window

The Find page provides full text search functionality that allows the user to search for any word or phrase in the Help file. This capability requires a full-text index file, which you can create when building the Help file, or which the user can create when using Help, as shown in Figure 12.16.

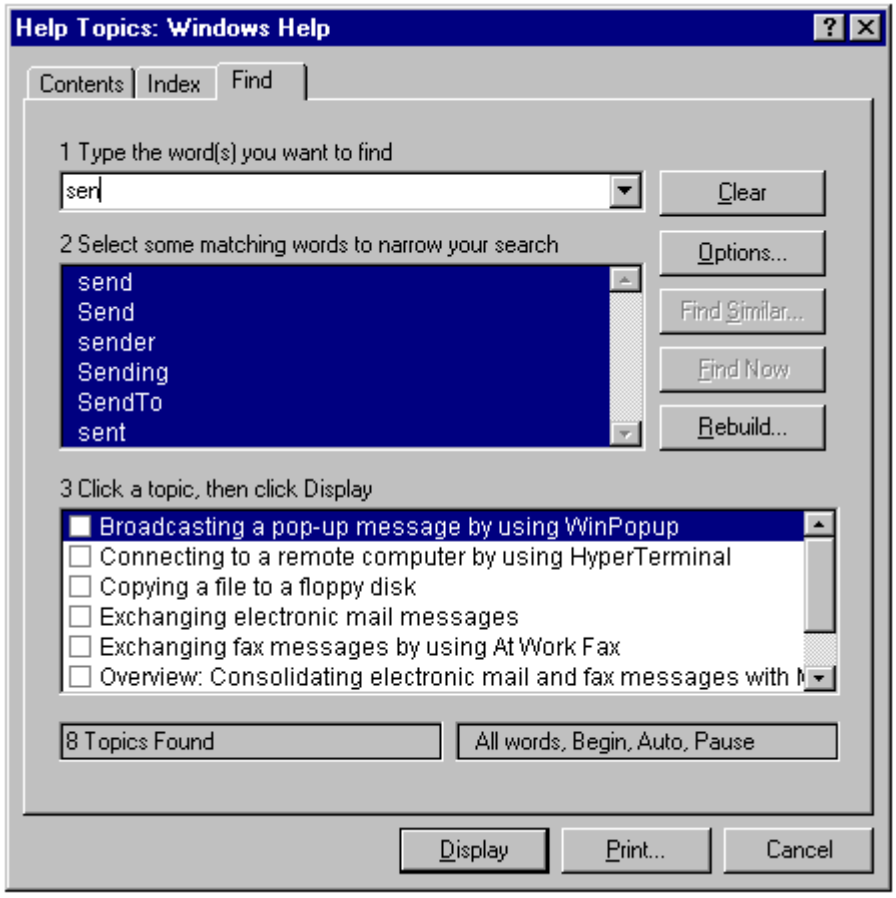


Figure 12.16 The Find page of the Help Topics browser

Guidelines for Writing Help Contents Entries

The entries listed on the Contents page are based on what you author in your Help files. Define them to allow the user to see the organizational relationship between topics. Make the topic titles you include for your software brief, but descriptive, and correspond to the actual topic titles.

Guidelines for Writing Help Index Keywords

Provide an effective keyword list to help users find the information they are looking for. When deciding what keywords to provide for your topics, consider the following categories:

- Words for a novice user
- Words for an advanced user
- Common synonyms of the words in the keyword list

- Words that describe the topic generally
- Words that describe the topic discretely

Wizards

A *wizard* is a special form of user assistance that automates a task through a dialog with the user. Wizards help the user accomplish tasks that can be complex and require experience. They also provide the interface for streamlining certain tasks.

Wizards may not always appear as an explicit part of the Help interface. You can provide access to them in a variety of ways, including toolbar buttons or even specific objects, such as templates.

For more information about template objects, see Chapter 5, “General Interaction Techniques.”

Wizard Buttons

At the bottom of the window, include the following command buttons that allow the user to navigate through the wizard.

Command	Action
<Back	Returns to the previous page. (Disables the button on the first page.)
Next>	Moves to the next page in the sequence, maintaining whatever settings t provides in previous pages.
Finish	Applies user-supplied or default settings from all pages and completes t
Cancel	Discards any user-supplied settings, terminates the process, and closes t wizard window.

Guidelines for Writing Text for Wizards

A wizard is a series of presentations or pages, displayed in a secondary window, that helps the user automate a task. The pages include controls that you define to gather input from the user; that input is then used to complete the task for the user. Wizards can automate almost any task, including creating new objects and formatting the presentation of a set of objects, such as a table or paragraph. They are especially useful for complex or infrequent tasks that the user may have difficulty learning or doing. However, use them to supplement, rather than replace, the user's direct ability to perform those tasks. The system provides support for creating wizards with the property sheet controls.

For more information about this control, see Chapter 7, "Menus, Controls, and Toolbars."

Optionally, you can define wizards as a series of secondary windows through which the user navigates. However, this can lead to increased modality and screen clutter, so using a single secondary window is recommended.

On the first page of a wizard, include a graphic in the left side of the window, as shown in Figure 12.17. The purpose of this graphic is to establish a reference point, or theme—such as a conceptual rendering, a snapshot of the area of the display that will be affected, or a preview of the result. On the top right portion of the wizard window, provide a short paragraph that welcomes the user to the wizard and explains what it does.

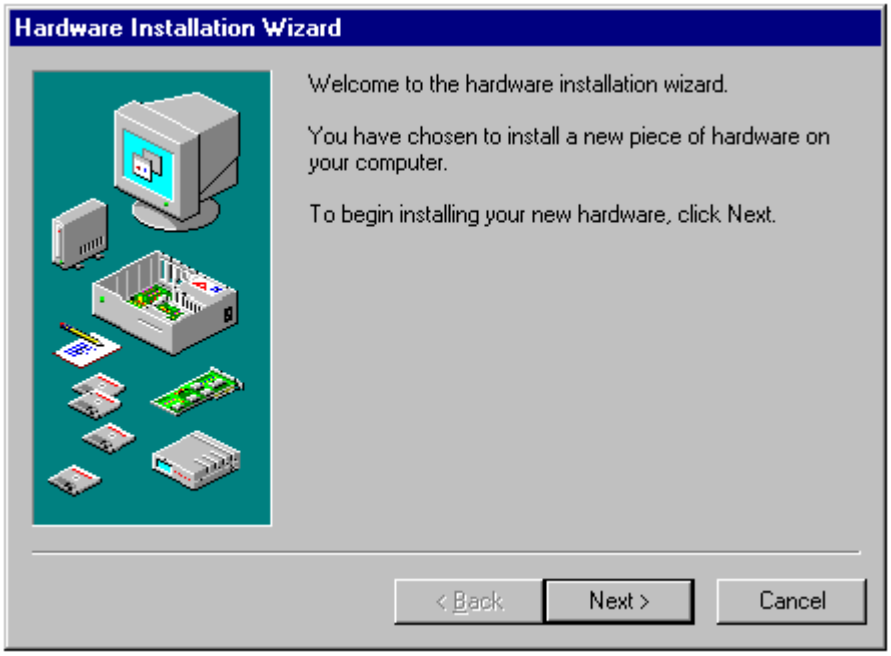


Figure 12.17 The introductory page of a wizard

On subsequent pages you can continue to include a graphic for consistency or, if space is critical, use the entire width of the window for displaying instructional text and controls for user input. When using graphics, include pictures that help illustrate the process, as shown in Figure 12.18. Include default values or settings for all controls where possible.

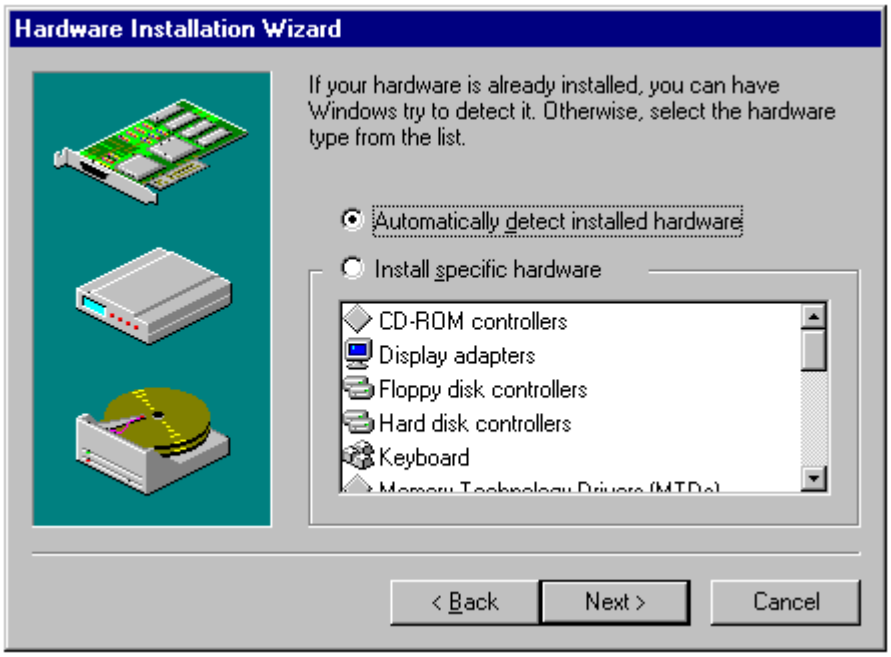


Figure 12.18 Input page for a wizard

You can include the Finish button at any point that the wizard can complete the task. For example, if you can provide reasonable defaults, you can even include the Finish button on the first page. This allows the user to step through the entire wizard or only the page on which they wish to provide input. On the last screen of the wizard, indicate to the user that the wizard is prepared to complete the task and instruct the user to click the Finish button to proceed.

Design your wizard pages to be easy to understand. It is important that users immediately understand what a wizard is about so they don't feel like they have to read it very carefully to understand what they have to answer. It is better to have more simple pages with fewer choices than complex pages with too many options or text. Similarly, while you can include controls that display a secondary window, minimize their use to keep the wizard operation simple and direct.

Make certain that the design alternatives offered by your wizard provide the user with positive results. You can use the context, such as the selection, to determine what options may be reasonable to provide. In addition, make certain that it is obvious how the user can proceed when the wizard has completed its process. This may be accomplished by the text you include on the last page of the wizard.

Guidelines for Writing Text for Wizard Pages

Use a conversational, rather than instructional, writing style for the text you provide on the screens. The following guidelines can be used to assist you in writing the textual information:

- Use words like "you" and "your."

- Start most questions with phrases like "Which option do you want..." or "Would you like...." Users respond better to questions that enable them to do a task than being told what to do. For example, "Which layout do you want?" works better in wizards than "Choose a layout."
- Use contractions and short, common words. In some cases, it may be acceptable to use slang, but consider localization when doing so.
- Avoid using technical terminology that may be confusing to a novice user.
- Try to use as few words as possible. For example, the question "Which style do you want for this newsletter?" could be written simply as "Which style do you want?"
- Keep the writing clear, concise, and simple, but not condescending.

CHAPTER 13

Visual Design

What we see influences how we feel and what we understand. Visual information communicates nonverbally, but very powerfully. It can include emotional cues that motivate, direct, or distract. The visual design of Microsoft Windows reflects an objective for making the interface easy to learn and use by effectively using visual communication and aesthetics. This chapter covers the visual and graphic design principles and guidelines that you can apply to the interface design of your Windows-based applications.

Visual Communication

Effective visual design serves a greater purpose than decoration; it is an important tool for communication. How you organize information on the screen can make the difference between a design that communicates a message and one that leaves a user feeling puzzled or overwhelmed.

Even the best product functionality can suffer if it doesn't have a well-designed, effective visual presentation. If you are not trained in visual or information design, it is a good idea to work with a designer who has education and experience in this field and include that person as a member of the design team early in the development process. Good graphic designers provide a perspective concerning how to take the best advantage of the screen and how to use effectively the concepts of shape, color, contrast, focus, and composition. Moreover, graphic designers understand how to design and organize information, and the effects of fonts and color on perception.

Composition and Organization

We organize what we read and how we think about information by grouping it spatially. We read a screen in the same way we read other forms of information. The eye is always attracted to the colored elements before black and white, to isolated elements before elements in a group, and to graphics before text. We even read text by scanning the shapes of groups of letters. Consider the following principles when designing the organization and composition of visual elements of your interface: hierarchy of information, focus and emphasis, structure and balance, relationship of elements, readability and flow, and unity of integration.

Hierarchy of Information

The principle of hierarchy of information addresses the placement of information based on its relative importance to other visual elements. The outcome of this ordering affects all of the other composition and organization principles, and determines what information a user sees first and what a user is encouraged to do first. To further consider this principle, ask these questions:

- What information is most important to a user?

  In other words, what are the priorities of a user when encountering your application's interface. For example, the most important priority may be to create or find a document.

- What does a user want or need to do first, second, third, and so on?

  Will your ordering of information support or complicate a user's progression through the interface?

- What should a user see on the screen first, second, third, and so on?

  What a user sees first should match the user's priorities when possible, but can be affected by the elements you want to emphasize.

Focus and Emphasis

The related principle of focus and emphasis guides you in the placement of priority items. Determining focus involves identifying the central idea, or the focal point, for activity. Determine emphasis by choosing the element that must be prominent and isolating it from the other elements or making it stand out in other ways.

Where the user looks first for information is an important consideration in the implementation of this principle. Culture and interface design decisions can govern this principle. People in western cultures, for example, look at the upper left corner of the screen or window for the most important information. So, it makes sense to put a top-priority item there, giving it emphasis.

Structure and Balance

The principle of structure and balance is one of the most important visual design principles. Without an underlying structure and a balance of those elements, there is a lack of order and meaning and this affects all other parts of the visual design. More importantly, a lack of structure and balance makes it more difficult for the user to clearly understand the interface.

Relationship of Elements

The principle of relationship of elements is important in reinforcing the previous principles. The placement of a visual element can help communicate a specific relationship of the elements of which it is a part. For example, if a button in a dialog box affects the content of a list box, there should be a spatial relationship between the two elements. This helps the user to clearly and quickly make the connection just by looking at the placement.

Readability and Flow

This principle calls for ideas to be communicated directly and simply, with minimal visual interference. Readability and flow can determine the usability of a dialog box or other interface component. When designing the layout of a window, consider the following:

- Could this idea or concept be presented in a simpler manner?
- Can the user easily step through the interface as designed?
- Do all the elements have a reason for being there?

Unity and Integration

The last principle, unity and integration, reflects how to evaluate a given design in relationship to its larger environment. When an application's interface is visually unified with the general interface of Windows, the user finds it easier to use because it offers a consistent and predictable work environment. To implement this principle, consider the following:

- How do all of the different parts of the screen work together visually?
- How does the visual design of the application relate to the system's interface or other applications with which it is used?

Color

Color is a very important aesthetic property in the visual interface. Because color has attractive qualities, use it to identify elements in the interface to which you want to draw the user’s attention—for example, the current selection. Color also has an associative aspect; we often assume there is a relationship between items of the same color. Color also carries with it emotional or psychological qualities. For example, colors are often categorized as cool or warm.

When used indiscriminately, color can have a negative or distracting effect. It can affect not only the user’s reaction to your software but also productivity, by making it difficult to focus on a task.

In addition, there are a few more things to consider about using color:

- While you can use color to show relatedness or grouping, associating a color with a particular meaning is not always obvious or easily learned.
- Color is a very subjective property. Everyone has different tastes in color. What is pleasing to you may be distasteful to someone else.
- Some percentage of your users may work with equipment that only supports monochrome presentation.
- Interpretation of color can vary by culture. Even within a single culture, individual associations with color can differ.
- Some percentage of the population may have color-identification problems. For example, about 10 percent of the adult male population have some form of color confusion.

The following sections summarize guidelines for using color: color as a secondary form of information, use of a limited set of colors, allowing the option to change colors.

Color as a Secondary Form of Information

Use color as an additive, redundant, or enhanced form of information. Avoid relying on color as the only means of expressing a particular value or function. Shape, pattern, location, and text labels are other ways to distinguish information. It is also a good practice to design visuals in black and white or monochrome first, then add color.

Use of a Limited Set of Colors

While the human eye can distinguish millions of different colors, using too many usually results in visual clutter and can make it difficult for the user to discern the purpose of the color information. The colors you use should fit their purpose. Muted, subtle, complementary colors are usually better than bright, highly saturated ones, unless you are really looking for a carnival-like appearance where bright colors compete for the user's attention.

Color also affects color. Adjacent or background colors affect the perceived brightness or shade of a particular color. A neutral color (for example, light gray) is often the best background color. Opposite colors, such as red and green, can make it difficult for the eye to focus. Dark colors tend to recede in the visual space, while light colors come forward.

Options to Change Colors

Because color is a subjective, personal preference, allow the user to change colors where possible. For interface elements, Windows provides standard system interfaces and color schemes. If you base your software on these system properties, you can avoid including additional controls, plus your visual elements are more likely to coordinate effectively when the user changes system colors. This is particularly important if you are designing your own controls or screen elements to match the style reflected in the system.

When providing your own interface for changing colors, consider the complexity of the task and skill of the user. It may be more helpful if you provide palettes, or limited sets of colors, that work well together rather than providing the entire spectrum. You can always supplement the palette with an interface that allows the user to add or change a color in the palette.

Fonts

Fonts have many functions in addition to providing letterforms for reading. Like other visual elements, fonts organize information or create a particular mood. By varying the size and weight of a font, we see text as more or less important and perceive the order in which it should be read.

By the very nature of the computer screen, fonts are generally less legible online than on a printed page. Avoid italic and serif fonts; these are often hard to read, especially at low resolutions. Figure 13.1 shows various font choices.

Figure 13.1 Effective and ineffective font choices

Limit the number of fonts and styles you use in your software's interface. Using too many fonts usually results in visual clutter.

Wherever possible, use the standard system font for common interface elements. This provides visual consistency between your interface and the system's interface and also makes your interface more easily scaleable. Because many interface elements can be customized by the user, check the system settings for the default system font and set the fonts in your interface accordingly.

For more information about system font settings, see the section, "Layout," later in this chapter.

Dimensionality

Many elements in the Windows interface use perspective, highlighting, and shading to provide a three-dimensional appearance. This emphasizes function and provides real-world feedback to the user's actions. For example, command buttons have the same appearance as real buttons do. This provides the user with natural visual cues and the ability to discriminate between different types of information.

Windows bases its three-dimensional effects on a common theoretical light source, the conceptual direction that light would be coming from to produce the lighting and shadow effects used in the interface. The light source in Windows comes from the upper left.

In general, three-dimensional objects should look like their real-world counterparts. Introduce enough detail to communicate the real-world association and no more. In addition, avoid using three-dimensional effects for an element that is not interactive.

Design of Visual Elements

All visual elements influence one another. Effective visual design depends on context. In a graphical user interface, a graphic element and its function are completely interrelated. A graphical interface needs to function intuitively—it needs to look the way it works and work the way it looks.

Basic Border Styles

Windows provides a unified visual design for building visual components based on the border styles, as shown in Figure 13.2.

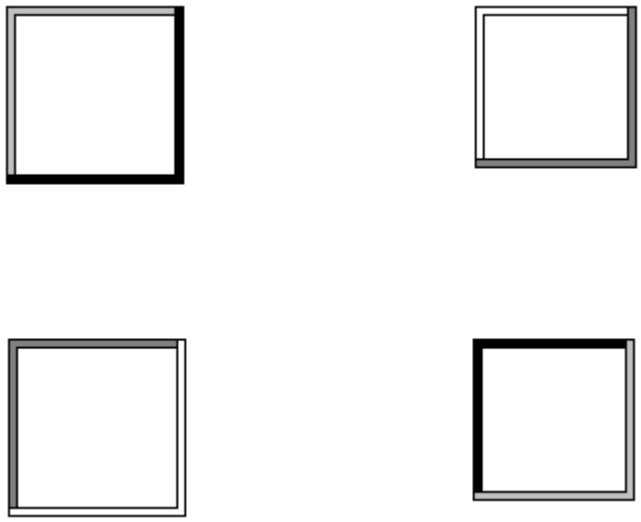


Figure 13.2 Basic border styles

Border style	Description
Raised outer border	Use a single-pixel width line in the button face color for its top and left edges and the window frame color for its bottom and right edges.
Raised inner border	Use a single-pixel width line in the button highlight color for its top and left edges and the button shadow color for its bottom and right edges.
Sunken outer border	Use a single-pixel width line in the button shadow color for its top and left border and the button highlight color for its bottom and right edges.
Sunken inner border	Use a single-pixel width line in the window frame color for its top and left edges and the button face color for its bottom and right edges.

If you use standard Windows controls and windows, these border styles are automatically supplied for your application. If you create your own controls, your application should map the colors of those controls to the appropriate system colors so that the controls fit in the overall design of the interface when the user changes the basic system colors.

The standard system color settings can be accessed using the **GetSysColor** function. The **DrawEdge** function automatically provides these border styles using the correct color settings.

For more information about the **GetSysColor** and **GetSysColor** functions, see the *Microsoft Win32 Programmer's Reference*.

Window Border Style

The borders of primary and secondary windows use the window border style, except for pop-up windows. Menus, scroll arrow buttons, and other situations where the background color may vary also use this border style. The border style is composed of the raised outer and raised inner basic border styles, as shown in Figure 13.3.

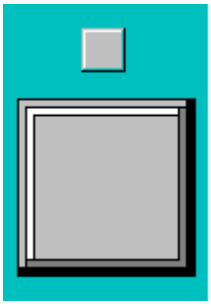


Figure 13.3 Window border style

Button Border Styles

Command buttons use the button border style. The button border style uses a variation of the basic border styles where the colors of the top and left outer and inner borders are swapped when combining the borders, as shown in Figure 13.4.

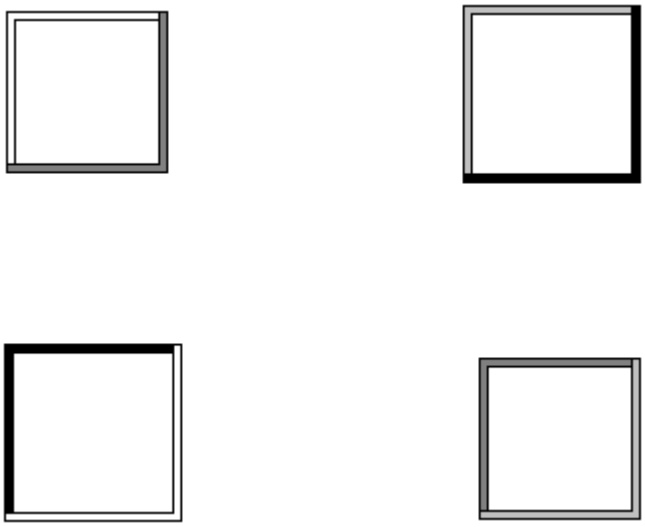


Figure 13.4 Button border styles

The normal button appearance combines the raised outer and raised inner button borders. When the user presses the button, the sunken outer and sunken inner button border styles are used. The button down border style (shown in Figure 13.5) is also used for read-only areas (for example, read-only text boxes).

Figure 13.5 Button up and button down border styles

Field Border Style

Text boxes (text boxes that are editable), check boxes, drop-down combo boxes, drop-down list boxes, spin boxes, list boxes, and wells use the field border style, as shown in Figure 13.6. You can also use the style to define the work area within a window. It uses the sunken outer and sunken inner basic border styles.

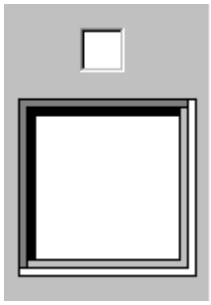


Figure 13.6 The field border style

For most controls, the interior of the field uses the button highlight color. However, in wells, the color may vary based on how the field is used or what is placed in the field, such as a pattern or color sample.

Status Field Border Style

Status fields use the status field border style, as shown in Figure 13.7. This style uses only the sunken outer basic border style.

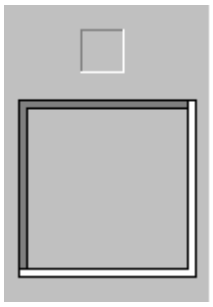


Figure 13.7 The status field border style

Grouping Border Style

Group boxes and menu separators use the grouping border style, as shown in Figure 13.8. The style uses the sunken outer and raised inner basic border styles.

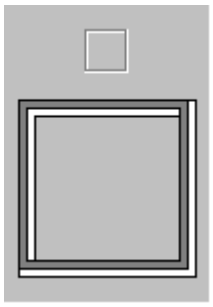


Figure 13.8 The group border style

Visual States for Controls

The visual design of controls must include the various states supported by the control. If you use standard Windows controls, Windows automatically provides specific appearances for these states. If you design your own controls, use the information in the previous section for the appropriate border style and information in the following sections to make your controls consistent with standard Windows controls.

For more information about standard control behavior and appearance, see the *Microsoft Win32 Programmer's Reference*.

Pressed Appearance

When the user presses a control, it provides visual feedback on the down transition of the mouse button. (For the pen, the feedback provided is for when the pen touches the screen and, for the keyboard, upon the down transition of the key.)

For standard Windows check boxes and option buttons, the background of the button field is drawn using the button face color, as shown in Figure 13.9.

Figure 13.9 Pressed appearance for check boxes and option buttons

For command buttons, the button-down border style is used and the button label moves down and to the right by one pixel, as shown in Figure 13.10.

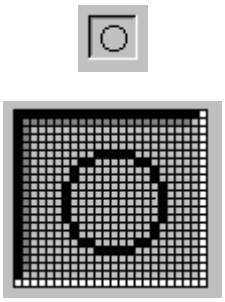


Figure 13.10 Pressed appearance for a command button

Option-Set Appearance

When using buttons to indicate when its associated value or state applies or is currently set, the controls provide an *option-set appearance*. The option-set appearance is used upon the up transition of the mouse button or pen tip, and the down transition of a key. It is visually distinct from the pressed appearance.

Standard check boxes and option buttons provide a special visual indicator when the option corresponding to that control is set. A check box uses a check mark, and an option button uses a large dot that appears inside the button, as shown in Figure 13.11.

Figure 13.11 Option-set appearance for check boxes and option buttons

When using command buttons to represent properties or other state information, the button face reflects when the option is set. The button continues to use the button-down border style, but a checkerboard pattern using the color of the button face and button highlight is displayed on the background of the button, as shown in Figure 13.12. The glyph on the button does not otherwise change from the pressed appearance.

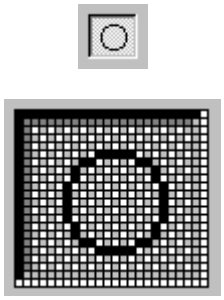


Figure 13.12 Option-set appearance for a command button

For well controls (shown in Figure 13.13), when a particular choice is set, place a border around the control, using the text color and the color of the button highlight.

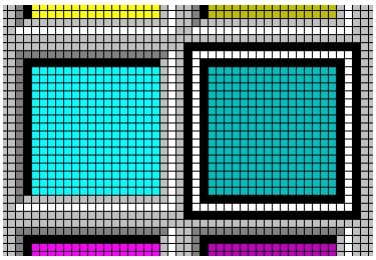


Figure 13.13 Option-set appearance for a well

Mixed-Value Appearance

When a control represents a property or other setting that reflects a set of objects where the values are different, the control is displayed with a *mixed-value* appearance (also referred to as indeterminate appearance), as shown in Figure 13.14.

For most standard controls, leave the field blank (that is, with no indication of a current set value) if it represents a mixed value. Standard check boxes support a special appearance for this state that draws a special mark in the box.

The system defines these states as BS_3STATE and BS_AUTO3STATE when using the **CreateWindow** and **CreateWindowEx** functions.

For more information about these constants and their functions, see the *Microsoft Win32 Programmer's Reference*.

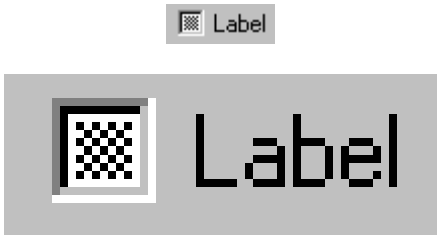


Figure 13.14 Mixed-value Appearance for a check box

For graphical command buttons, such as those used on toolbars, a checkerboard pattern using the button highlight color is drawn on the background of the button face, as shown in Figure 13.15.

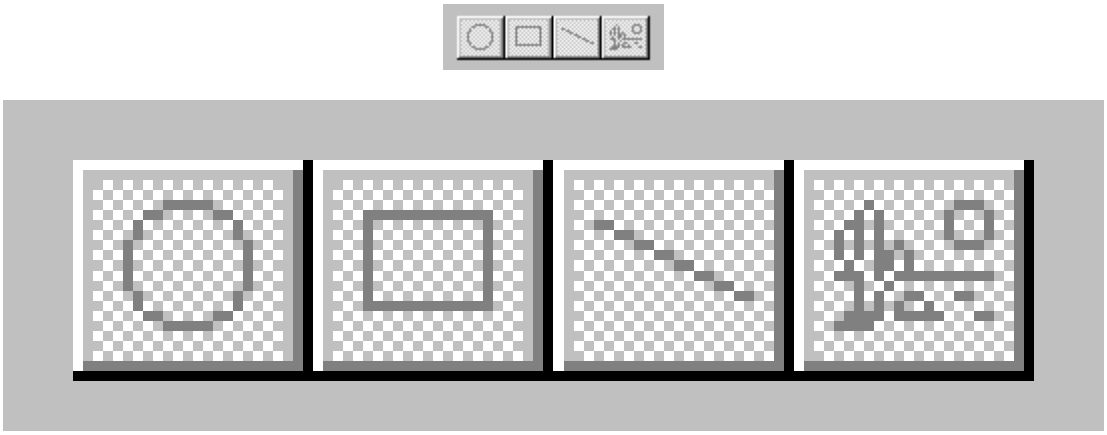


Figure 13.15 Mixed-value appearance for buttons

For button controls, when the user clicks the button, the property value or state is set. Clicking a second time clears the value. Typically, a third click returns the button to the mixed-value state.

Unavailable Appearance

When a control is unavailable (also referred to as disabled), its normal functionality is no longer available to the user (though it can still support access to contextual Help information) because the functionality represented does not apply or is inappropriate under the current circumstances. To reflect this state, the label of the control is rendered with a special engraved *unavailable appearance*, as shown in Figure 13.16.

Figure 13.16 Unavailable appearance for check boxes and option buttons

For graphical or textual buttons, create the engraved effect by drawing the label in the color of the button highlight and then overlaying it, at a small offset, with the image drawn in the color of the button shadow, as shown in Figure 13.17.

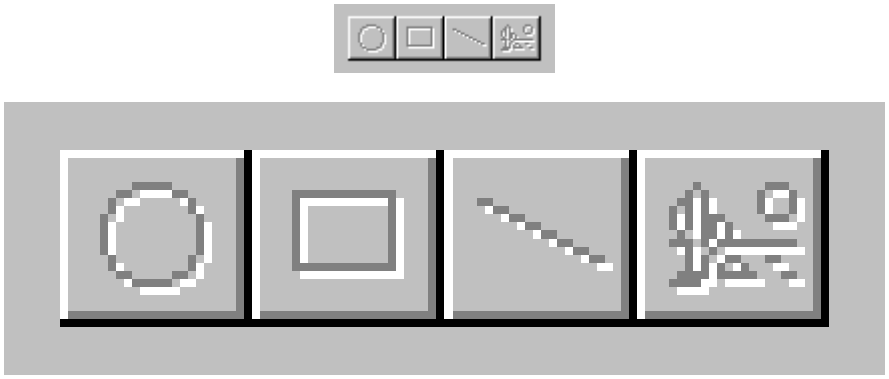


Figure 13.17 Unavailable appearance for buttons

If a check box or option button is set, but the control is unavailable, then the control's label is displayed with an unavailable appearance, and its mark appears in the button shadow color, as shown in Figure 13.18.

Figure 13.18 Unavailable appearance for check boxes and option buttons (when set)

If a graphical button needs to reflect both the set and unavailable appearance (as shown in Figure 13.19), omit the background checkerboard pattern and use the unavailable appearance for the button’s label.

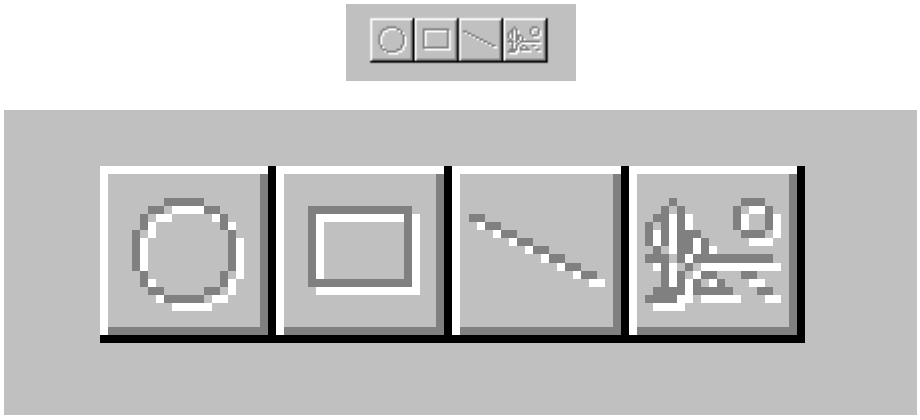


Figure 13.19 Unavailable and option-set appearance for buttons

Input Focus Appearance

You can provide a visual indication so the user knows where the input focus is. For text boxes, the system provides a blinking cursor, or insertion point. For other controls a dotted outline is drawn around the control or the control's label, as shown in Figure 13.20.

Figure 13.20 Example of Input focus in a control

The system provides support for drawing the dotted outline input focus indicator using the **DrawFocusRect** function. It is supported automatically for standard controls. To use it with your own custom controls, specify the rectangle to allow at least one pixel of space around the extent of the control. If the input focus indicator would be too intrusive, as an option, you can include it around the label for the control.

For more information about the **DrawFocusRect** function, see the *Microsoft Win32 Programmer's Reference*.

Flat Appearance

When you nest controls inside of a scrollable region or control, avoid using a three-dimensional appearance because it may not work effectively against the background. Instead, use the flat appearance style, as shown in Figure 13.21.

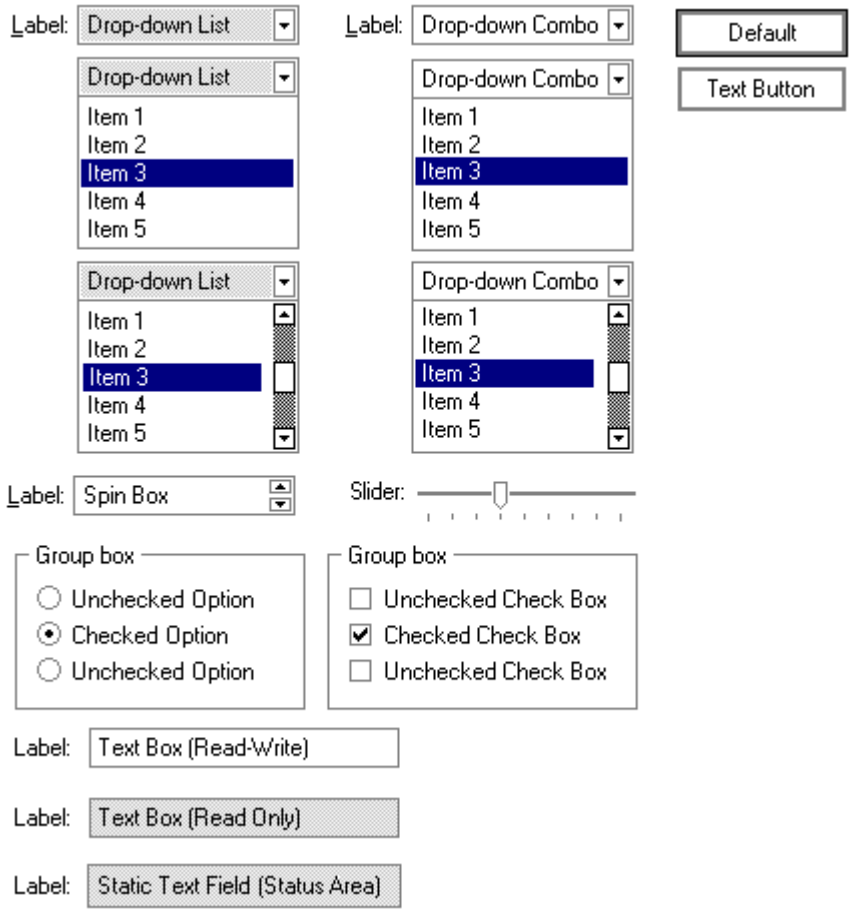


Figure 13.21 Flat appearance for standard controls

The system provides this appearance for standard Windows controls. Use the **DrawFrameControl** function with the DFCS_FLAT value. If you draw your own controls, use the **DrawEdge** function with the BF_FLAT value.

For more information about the **DrawFrameControl** and **DrawEdge** functions, see the *Microsoft Win32 Programmer's Reference*.

Layout

Size, spacing, and placement of information are critical in creating a visually consistent and predictable environment. Visual structure is also important for communicating the purpose of the elements displayed in a window. In general, follow the layout conventions for how information is read. In western countries, this means left-to-right, top-to-bottom, with the most important information located in the upper left corner.

Font and Size

The default system font is a key metric in the presentation of visual information. The default font used for interface elements in Windows (U.S. release) is MS Sans Serif for 8-point. Menu bar titles, menu items, control labels, and other interface text all use 8-point MS Sans Serif. Title bar text also uses the 8-point MS Sans Serif bold font, as shown in Figure 13.22. However, because the user can change the system font, make certain you check this setting and adjust the presentation of your interface appropriately.

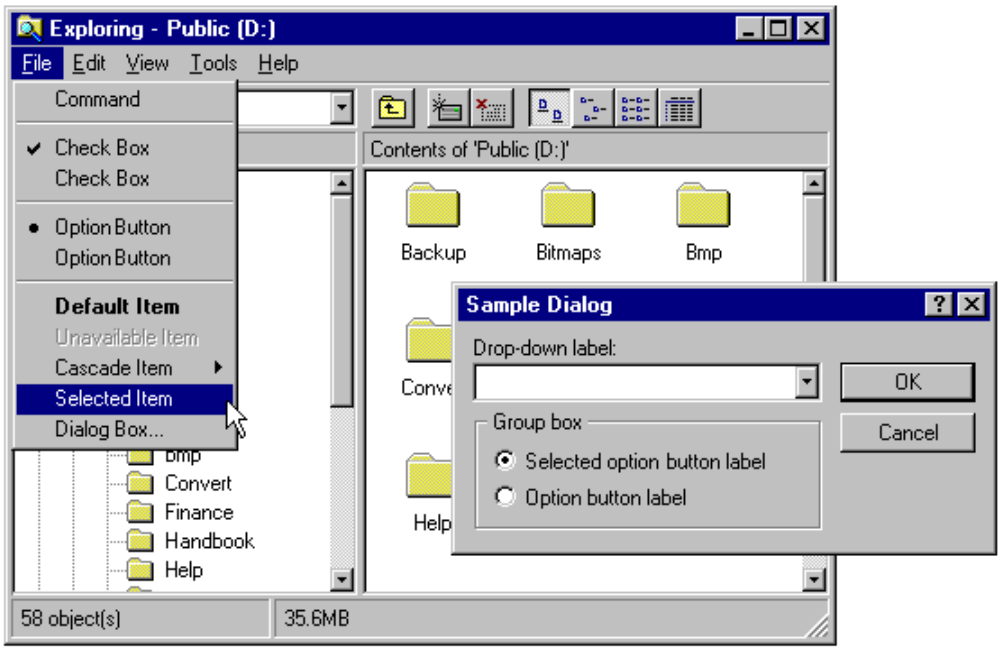


Figure 13.22 Font usage in Windows – MS Sans Serif

The system also provides settings for the font and size of many system components including title bar height, menu bar height, border width, title bar button height, icon title text, and scroll bar height and width. When designing your window layouts, take these variables into consideration so that your interface will scale appropriately. In addition, use these standard system settings to determine the size of your custom interface elements.

You can access the system settings for many standard window interface elements using the **GetSystemMetrics** function. You can access the current system font settings using **SystemParametersInfo** (primary window fonts) and **GetStockObject** (dialog box fonts).

For more information about the **GetSystemMetrics**, **SystemParametersInfo** and **GetStockObject** functions, see the *Microsoft Win32 Programmer's Reference*.

The default size for most single-line controls is 14 dialog base units. A *dialog base unit* is a device-independent measure to use for layout. One horizontal unit is equal to one-fourth of the average character width for the current system font. One vertical unit is equal to one-eighth of an average character height for the current system font. Figure 13.23 shows the recommended spacing.

Your application can retrieve the number of pixels per base unit for the current display using the **GetDialogBaseUnits** function.

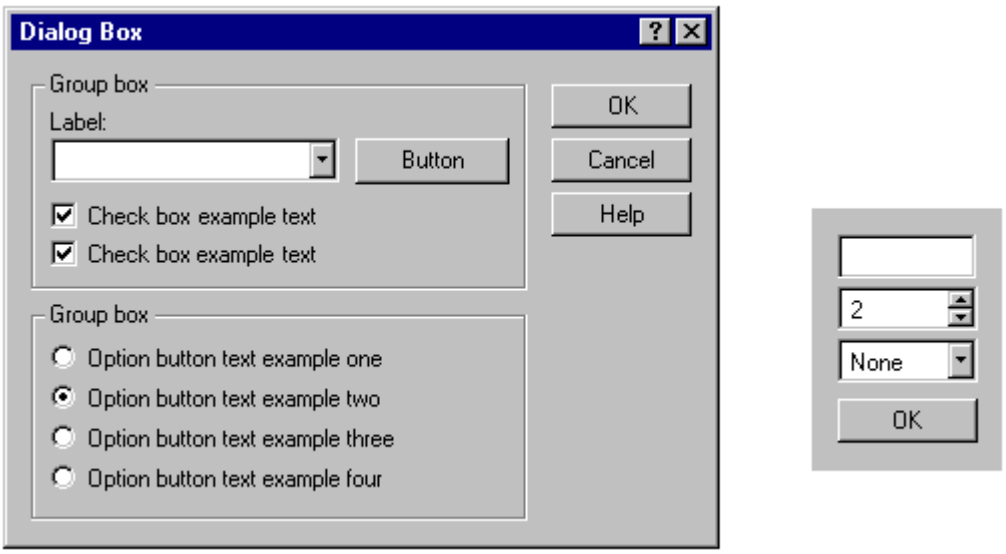


Figure 13.23 Recommended layout of controls and text

If a menu item represents a default command, the text is bold. Default command buttons use a bold outline around the button. In general, use nonbold text in your windows. Use bold text only when you want to call attention to an area or create a visual hierarchy.

The recommended maximum size for secondary windows, such as dialog boxes and property sheets, is 234 dialog base units wide by 263 dialog base units high (415 pixels by 452 pixels). Maintaining this size keeps the window from becoming too large to display at some resolutions, and provides reasonable space to display supportive information, such as Help windows that apply to the dialog box or property sheet.

Toolbar buttons should be 24 pixels wide by 22 pixels high on VGA resolution. This includes the border. For greater resolutions, you can proportionally size the button to be the same height as a text box control. This allows the button to maintain its proportion with respect to other controls. The glyph used on the button should be 16 pixels wide by 16 pixels high for VGA resolution. For greater resolutions, stretch the glyph when the button is an even multiple of the VGA resolution; that is, if the button is twice as big, then double the glyph. Your application can also supply alternative images for higher resolutions. This can be preferable, because it provides better visual results. Center the glyph on the button’s face.

Toolbar buttons generally have only graphical labels and no accompanying textual label. You can use a tooltip to provide the name of the button.

For more information about tooltips, see Chapter 7, "Menus, Controls, and Toolbars," and Chapter 12, "User Assistance."

Capitalization

When defining command names in menus and command buttons, use conventional title capitalization. Capitalize the first letter in each word unless it is an article or preposition not occurring at the beginning or end of the name, or unless the word's conventional usage is not capitalized. For example:

Insert Object
Paste Link
Save As
Go To
Always on Top
By Name

Default names provided for title bar text or icon names also follow this convention. Of course, if the user supplies a name for an object, display the name as the user specifies it, regardless of case.

Field labels, such as those used for option buttons, check boxes, text boxes, group boxes, and page tabs, should use sentence-style capitalization. Capitalize the first letter of the initial word and any words that are normally capitalized. For example:

Extended (XMS) memory
Working directory
Print to
Find whole words only

Grouping and Spacing

Group related components together. You can use group box controls or spacing. Leave at least three dialog base units between controls. You can also use color to visually group objects, but this often involves design tradeoffs. Maintain consistent spacing (seven dialog base units is recommended) from the edge of the window. Use spacing between groups within the window.

For more information about the use of color, see the section, "Color," earlier in this chapter.

Position controls in a toolbar so that there is at least a window's border width from the edges of the toolbar. You can apply the same measure for spacing between buttons, unless you want to align a set of related buttons adjacently. Use adjacent alignment for buttons that form an exclusive choice set. That is, when using buttons like a set of option buttons, align them without any spacing between them.

Alignment

When information is positioned vertically, align fields by their left edges (in western countries). This usually makes it easier for the user to scan the information. Text labels are usually left aligned and placed above or to the left of the areas to which they apply. When placing text labels to the left of text box controls, align the height of the text with text displayed in the text box.

Button Placement

Stack the main command buttons in a secondary window in the upper right corner or in a row along the bottom, as shown in Figure 13.24. If there is a default button, it is typically the first button in the set. Place OK and Cancel buttons next to each other. The last button is a Help button (if supported). If there is no OK button, but other action buttons, it is best to place the Cancel button at the end of a set of action buttons, but before a Help button. If a particular command button applies only to a particular field, you may place it grouped with that field.

For more information about button placement in secondary windows, see Chapter 8, "Secondary Windows."

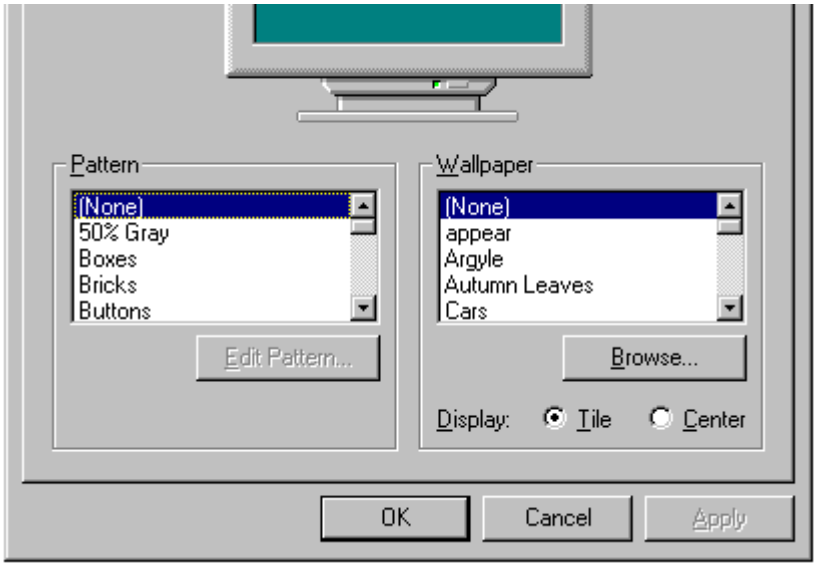


Figure 13.24 Layout of buttons

For easy readability, make buttons a consistent length. However, if maintaining this consistency greatly expands the space required by a set of buttons, it may be reasonable to have one button larger than the rest.

Placement of command buttons (or other controls) within a tabbed page implies the application of only the transactions on that page. If command buttons are placed within the window, but not on the tabbed page, they apply to the entire window.

Design of Graphic Images

When designing pictorial representations of objects, whether they are icons or graphical buttons, begin by defining the icon's purpose and its use. Brainstorm about possible ideas, considering real-world metaphors. It is often difficult to design icons that define operations or processes—activities that rely on verbs. Use nouns instead. For example, scissors can represent the idea of Cut.

Draw your ideas using an icon-editing utility or pixel (bitmapped) drawing package. Drawing them directly on the screen provides immediate feedback about their appearance.

It is a good idea to begin the design in black and white. Consider color as an enhancing property. Also, test your images on different backgrounds. They may not always be seen against white or gray backgrounds.

An illustrative style tends to communicate metaphorical concepts more effectively than abstract symbols. However, in designing an image based on a real-world object, use only the amount of detail that is really necessary for user recognition and recall. Where possible and appropriate, use perspective and dimension (lighting and shadow) to better communicate the real-world representation, as shown in Figure 13.25.

Figure 13.25 Perspective and dimension improve graphics

Consistency is also important in the design of graphic images. As with other interface elements, design images assuming a light source from the upper left. In addition, make certain the scale (size) and orientation of your icons are consistent with the other objects to which they are related and fit well within the working environment.

You may want to include a technique called anti-aliasing. *Anti-aliasing* involves adding colored pixels to smooth the jagged edges of a graphic. However, do not use anti-aliasing on the outside edge of an icon as the contrasting pixels may look jagged or fuzzy on varying backgrounds.

Finally, remember to consider the potential cultural impact of your graphics. What may have a certain meaning in one country or culture may have unforeseen meanings in another. It is best to avoid letters or words, if possible, as this may make the graphics difficult to apply for other cultures.

For more information about designing for international audiences, see Chapter 14, “Special Design Considerations.”

Icon Design

Icons are used throughout the interface to represent objects or tasks. Because icons represent your software’s objects, it is important not only to supply effective icons, but to design them to effectively communicate their purpose.

When designing icons, design them as a set, considering their relationship to each other and to the user's tasks. Do several sketches or designs and test them for usability.

Sizes and Types

Supply icons for your application in all standard sizes: 16- by 16-pixel, 32- by 32-pixel, and 48- by 48- pixel, as shown in Figure 13.26.

Figure 13.26 Three sizes of icons

Define icons not only for your application executable file, but also for all data file types supported by your application, as shown in Figure 13.27.

Figure 13.27 Application and supported document icons

Icons for documents and data files should be distinct from the application's icon. Include some common element of the application's icon, but focus on making the document icon recognizable and representative of the file's content.

Register the icons you supply in the system registry. If your software does not register any icons, the system automatically provides one, as shown in Figure 13.28. However, it is unlikely to be as detailed or distinctive as one you can supply.

For more information about registering your icons, see Chapter 10, "Integrating with the System."

Figure 13.28 System-generated icon that is not registered

Icon Style

When designing your icons, use a common style across all icons. Repeat common characteristics, but avoid repeating unrelated elements.

Some icon style recommendations have been slightly modified from those used in Windows 3.1. The new recommendations reflect the same light source as the system controls. Black outlines have also been eliminated to reduce visual clutter, as shown in Figure 13.29.

Figure 13.29 Revised icon style

User Recognition and Recollection

User recognition and recollection are two important factors to consider in icon design. Recognition means that the icon is identifiable by the user and easily associated with a particular object. Support user recognition by using effective metaphors. Use real-world objects to represent abstract ideas so that the user can draw from previous learning and experiences. Exploit the user's knowledge of the world and allude to the familiar.

To facilitate recollection, design your icons to be simple and distinct. Applying the icon consistently also helps build recollection; therefore, design your small icons to be as similar as possible to their larger counterparts. It is generally best to try to preserve general shape and any distinctive detail. 48- by 48-pixel icons can be rendered in 256 colors. This allows very realistic-looking icons, but focus on simplicity and careful use of color. If your software is targeted at computers that can only display 256 colors, make certain you only use colors from the system's standard 256-color palette. If you aim at computers configured for 65,000 colors, you can use any combination of colors.

Pointer Design

You can use the pointer design to help the user identify objects and provide feedback about certain conditions or states. However, use pointer changes conservatively so that the user is not distracted by excessive flashing of multiple pointer changes while traversing the screen. One way to handle this is to use a time-out before making noncritical pointer changes.

When you use a pointer to provide feedback, use it only over areas where that state applies. For example, when using the hourglass pointer to indicate that a window is temporarily noninteractive, if the pointer moves over a window that is interactive, change it to its appropriate interactive image.

Pointer feedback may not always be sufficient. For example, for processes that last longer than a few seconds, it is better to use a progress indicator that indicates progressive status, elapsed time, estimated completion time, or some combination of these to provide more information about the state of the operation.

Use a pointer that best fits the context of the activity. The I-beam pointer is best used to select text. The normal arrow pointer works best for most drag and drop operations, modified when appropriate to indicate copy and link operations.

For more information about some of the common pointers, see Chapter 4, "Input Basics." For information about displaying pointers for drag and drop operations, see Chapter 5, "General Interaction Techniques."

The location for the hot spot of a pointer (shown in Figure 13.30) is important for helping the user target an object. The pointer's design should make the location of the hot spot intuitive. For example, for a cross-hair pointer, the implied hot spot is the intersection of the lines.

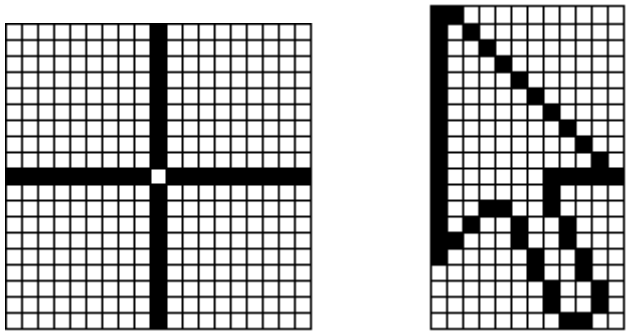


Figure 13.30 Pointer hot spots

Animating a pointer can be a very effective way of communicating information. However, remember that the goal is to provide feedback, not to distract the user. In addition, pointer animation should not restrict the user's ability to interact with the interface.

Selection Appearance

When the user selects an item, provide visual feedback to enable the user to distinguish it from items that are not selected. Selection appearance generally depends on the object and the context in which the selection appears.

Highlighting

For many types of object, you can display the background or some distinguishing part of the object (for example, a resizing handle) using the system highlight color. Figure 13.31 shows examples of selection appearances.

The system settings for interface colors such as the highlight color, COLOR_HIGHLIGHT, can be accessed using the **GetSysColor** function. For more information about this function, see the *Microsoft Win32 Programmer's Reference*.

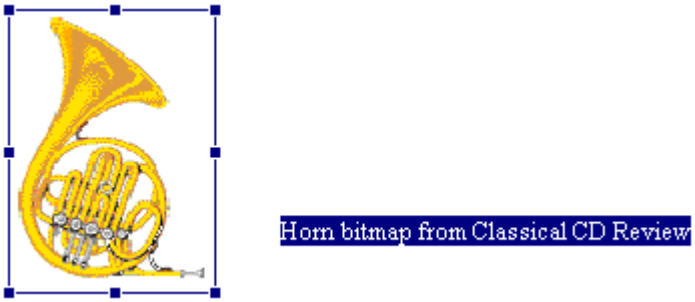


Figure 13.31 Examples of selection appearance

Display an object with selection appearance as the user performs a selection operation. For example, display selection appearance when the user presses the mouse button.

For more information about selection techniques, see Chapter 4, “Input Basics.”

It is best to display the selection appearance only for the scope, area, or level (window or pane) that is active. This helps the user recognize which selection currently applies and the extent of the scope of that selection. Therefore, avoid displaying selections in inactive windows or panes, or at nested levels.

However, in other contexts it may still be appropriate to display selection appearance simultaneously in multiple contexts. For example, when the user selects an object and then selects a menu item to apply to that object, selection appearance is always displayed for both items because it is clear where the user is directing the input. In cases such as in a hierarchical selection, where you need to show simultaneous selection, but with the secondary selection distinguished from the active selection, consider drawing an outline in the selection highlight color around the secondary selection or using some similar variant of the standard selection highlight technique.

Similarly, in a secondary window, it may be appropriate to display selection highlighting when the highlight is also being used to reflect the setting for a control. For example, in list boxes, highlighting often indicates a current setting. In cases like this, provide an input focus indication as well so the user can distinguish when input is being directed to another control in the window; you can also use check marks instead of highlighting to indicate the setting.

Handles

Handles provide access to operations for an object, but they can also indicate selection for some kinds of objects. The typical handle is a solid, filled square box that appears on the edge of the object. The handle is "hollow" when the handle indicates selection, but is not a control point by which the object may be manipulated. Figure 13.32 shows a solid and a hollow handle.

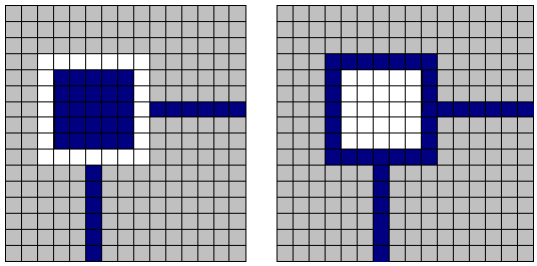


Figure 13.32 Normal and hollow handles

Base the default size of a handle on the current system settings for window border and edge metrics so that your handles are appropriately sized when the user explicitly changes window border widths or to accommodate higher resolutions. Similarly, base the colors you use to draw handles on system color metrics so that when the user changes the default system colors, handles change appropriately. Use the system highlighted text color for the border of normal handles and the selection highlight for the handle's fill color. For hollow handles use the opposite: selection highlight color for the border and highlighted text color for the fill color.

The system settings for window border and edge metrics can be accessed using the **GetSystemMetrics** function. For more information about this function, see the *Microsoft Win32 Programmer's Reference*.

Transfer Appearance

When the user drags an object to perform an operation, for example, move, copy, print, and so on, display a representation of the object that moves with the pointer. In general, do not simply change the pointer to be the object, as this may obscure the insertion point at some destinations. Instead, use a translucent or outline representation of the object that moves with the pointer, as shown in Figure 13.33.

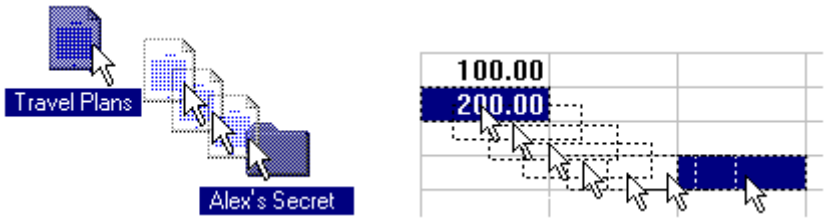


Figure 13.33 Translucent and outline representation (drag transfer)

You can create a translucent representation by using a checkerboard mask made up of 50 percent transparent pixels. When used together with the object’s normal appearance, this provides a representation that allows part of the destination to show through.

The presentation displayed is always defined by the destination. Use a representation that best communicates how the transferred object will be incorporated when the user completes the drag transfer. For example, if the object being dragged will be displayed as an icon, then display an icon as its representation. If, on the other hand, it will be incorporated as native content, then display an appropriate representation. You could display a graphics object as an outline or translucent image of its shape, a table cell as the outline of a rectangular box, and text selection as the first few characters of a selection with a transparent background.

Set the pointer image to be whichever pointer the target location uses for directly inserting information. For example, when dragging an object into normal text context, use the I-beam pointer. In addition, include the copy or link image at the bottom right of the pointer if that is the meaning for the operation.

For more information about transfer operations, see Chapter 5, "General Interaction Techniques."

Open Appearance

Open appearance is most commonly used for an OLE embedded object, but it can also apply in other situations where the user opens an object into its own window. To indicate that an object is "open," display the object in its container's window overlaid with a set of hatched (45 degree) lines drawn every four pixels, as shown in Figure 13.34.

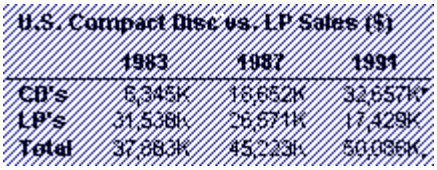


Figure 13.34 An object with opened appearance

For more information about the use of open appearance for OLE embedded objects, see Chapter 11, "Working with OLE Embedded and OLE Linked Objects."

Animation

Animation can be an effective way to communicate information. For example, it can illustrate the operation of a particular tool or reflect a particular state. It can also be used to include an element of fun in your interface. You can use animation effects for objects within a window and interface elements, such as icons, buttons, and pointers.

Effective animation involves many of the same design considerations as other graphics elements, particularly with respect to color and sound. Fluid animation requires presenting images at 16 (or more) frames per second.

When you add animation to your software, ensure that it does not affect the interactivity of the interface. Do not force the user to remain in a modal state to allow the completion of the animation. Unless animation is part of a process, make it interruptible by the user or independent of the user’s primary interaction.

Avoid gratuitous use of animation. When animation is used for decorative effect it can distract or annoy the user. You may want to provide the user with the option of turning off the animation or otherwise customizing the animation effects.

CHAPTER 14

Special Design Considerations

A well-designed Microsoft Windows application must consider other factors to appeal to the widest possible audience. This chapter covers special user interface design considerations, such as sound, accessibility, internationalization, and network computing.

Sound

You can incorporate sound as a part of an application in several ways—for example, music, speech, or sound effects. Such auditory information can take the following forms:

- A primary form of information, such as the composition of a particular piece of music or a voice message.
- An enhancement of the presentation of information but that is not required for the operation of the software.
- A notification or alerting of users to a particular condition.

Sound is an effective form of information and enhances the interface when appropriately used. Try to avoid using sound as the only means of conveying information. Some users may be hard-of-hearing or deaf. Others may work in a noisy environment or in a setting that requires that they disable sound or maintain it at a low volume. In addition, like color, sound is a very subjective part of the interface. As a result, sound is best incorporated as a redundant or secondary form of information, or supplemented with alternative forms of communication. For example, if a user turns off the sound, consider flashing the window's title bar, taskbar button, presenting a message box, or other means of bringing the user's attention to a particular situation. Even when sound is the primary form of information, you can supplement the audio portion by providing visual representation of the information that might otherwise be presented as audio output, such as captioning or animation.

The taskbar can also provide visual status or notification information. For more information about using the taskbar for this purpose, see Chapter 10, "Integrating with the System."

Always allow the user to customize sound support. Support the standard system interfaces for controlling volume and associating particular sounds with application-specific sound events. You can also register your own sound events for your application.

The system provides a global system setting, ShowSounds. The setting indicates that the user wants a visual representation of audio information. Your software should query the **GetSystemMetrics** function to check the status of this setting and provide captioning for the output of any speech or sounds. Captioning should provide as much information visually as is provided in the audible format. It is not necessary to caption ornamental sounds that do not convey useful information.

For more information about accessing the **GetSystemMetrics** function, the ShowSounds option, and the SoundSentry option, see the *Microsoft Win32 Programmer's Reference*.

Do not confuse the ShowSounds option with the system's SoundSentry option. When the user sets the SoundSentry option, the system automatically supplies a visual indication whenever a sound is produced.

Avoid relying on SoundSentry alone if the ShowSounds option is set because SoundSentry only provides rudimentary visual indications, such as flashing of the display or screen border, and it cannot convey the meaning of the sound to the user. The system provides SoundSentry primarily for applications that do not provide support for ShowSounds. The user sets either of these options with the Microsoft Windows Accessibility Options.

Note In Windows 95, SoundSentry only works for audio output directed through the internal PC speaker.

Accessibility

Accessibility means making your software usable and accessible to a wide range of users, including those with disabilities. A number of users require special accommodation because of temporary or permanent disabilities.

The issue of software accessibility in the home and workplace is becoming increasingly important. Nearly one in five Americans have some form of disability—and it is estimated that 30 million people in the U.S. alone have disabilities that may be affected by the design of your software. In addition, between seven and nine out of every ten major corporations employ people with disabilities who may need to use computer software as part of their jobs. As the population ages and more people become functionally limited, accessibility for users with disabilities will become increasingly important to the population as a whole. Legislation, such as the Americans with Disabilities Act, requires that most employers provide reasonable accommodation for workers with disabilities. Section 508 of the Rehabilitation Act is also bringing accessibility issues to the forefront in government businesses and organizations receiving government funding.

Designing software that is usable for people with disabilities does not have to be time consuming or expensive. However, it is much easier if you include this in the planning and design process rather than attempting to add it after the completion of the software. Following the principles and guidelines in this guide will help you design software for most users. Often recommendations, such as the conservative use of color or sound often benefit all users, not just those with disabilities. In addition, keep the following basic objectives in mind:

- Provide a customizable interface to accommodate a wide variety of user needs and preferences.
- Provide compatibility with accessibility utilities that users can install.
- Avoid creating unnecessary barriers that make your software difficult or inaccessible to certain types of users.

The following sections provide information on types of disabilities and additional recommendations about how to address the needs of customers with those disabilities.

Visual Disabilities

Visual disabilities range from slightly reduced visual acuity to total blindness. Those with reduced visual acuity may only require that your software support larger text and graphics. For example, the system provides scalable fonts and controls to increase the size of text and graphics. To accommodate users who are blind or have severe impairments, make your software compatible with the speech or Braille utilities.

Color blindness is another visual impairment that makes it difficult for users to distinguish between certain color combinations. This is one reason why color is not recommended as the only means of conveying information. Always use color as an additive or enhancing property.

Hearing Disabilities

Users who are deaf or hard-of-hearing are generally unable to detect or interpret auditory output at normal or maximum volume levels. Avoiding the use of auditory output as the only means of communicating information is the best way to support users with this disability. Instead, use audio output only as a redundant, additive property.

For more information about supporting sound, see the section "Sound" earlier in this chapter.

Physical Movement Disabilities

Some users have difficulty or are unable to perform certain physical tasks—for example, moving a mouse or simultaneously pressing two keys on the keyboard. Other individuals have a tendency to inadvertently strike multiple keys when targeting a single key. Consideration of physical ability is important not only for users with disabilities, but also for beginning users who need time to master all the motor skills necessary to interact with the interface. The best way to support users with physical movement disabilities is to provide good keyboard and mouse interfaces.

Speech or Language Disabilities

Users with language disabilities, such as dyslexia, find it difficult to read or write. Spell- or grammer-check utilities help children, users with writing impairments, and users with a different first language. Supporting accessibility tools and utilities designed for users who are blind can also help those with reading impairments. Most design issues affecting users with oral communication difficulties apply only to utilities specifically designed for speech input.

Cognitive Disabilities

Cognitive disabilities can take many forms, including perceptual differences and memory impairments. You can accommodate users with these disabilities by allowing them to modify or simplify your software's interface, such as supporting menu or dialog box customization. Similarly, using icons and graphics to illustrate objects and choices can be helpful for users with some types of cognitive impairments.

Seizure Disorders

Some users are sensitive to visual information that alternates its appearance or flashes at particular rates—often the greater the frequency, the greater the problem. However, there is no perfect flash rate. Therefore, base all modulating interfaces on the system's cursor blink rate. Because users can customize this value, a particular frequency can be avoided. If that is not practical, provide your own interface for changing the flash rate.

Types of Accessibility Aids

There are a number of accessibility aids to assist users with certain types of disabilities. To allow these users to effectively interact with your application, make certain it is compatible with these utilities. This section briefly describes the types of utilities and how they work.

One of the best ways to accommodate accessibility in your software's interface is to use standard Windows conventions wherever possible. Windows already provides a certain degree of customization for users and most accessibility aids work best with software that follows standard system conventions.

Screen Enlargement Utilities

Screen enlargers (also referred to as screen magnification utilities or large print programs) allow users to enlarge a portion of their screen. They effectively turn the computer monitor into a viewport showing only a portion of an enlarged virtual display. Users then use the mouse or keyboard to move this viewport to view different areas of the virtual display. Enlargers also attempt to track where users are working, following the input focus and the activation of windows, menus, and secondary windows, and can automatically move the viewport to the active area.

Screen Review Utilities

People who cannot use the visual information on the screen can interpret the information with the aid of a screen review utility (also referred to as a screen reader program or speech access utility). Screen review utilities take the displayed information on the screen and direct it through alternative media, such as synthesized speech or a refreshable Braille display. Because both of these media present only text information, the screen review utility must render other information on the screen as text; that is, determine the appropriate text labels or descriptions for graphical screen elements. They must also track users' activities to provide descriptive information about what the user is doing. These utilities often work by monitoring the system interfaces that support drawing on the screen. They build an off-screen database of the objects on the screen, their properties, and their spatial relationships. Some of this information is presented to users as the screen changes, and other information is maintained until users request it. Screen review utilities often include support for configuration files (also referred to as set files or profiles) for particular applications.

Voice Input Systems

Users who have difficulty typing can choose a voice input system (also referred to as a speech recognition program) to control software with their voice instead of a mouse and keyboard. Like screen reader utilities, voice input systems identify objects on the screen that users can manipulate. Users activate an object by speaking the label that identifies the object. Many of these utilities simulate keyboard interfaces, so if your software includes a keyboard interface, it can be adapted to take advantage of this form of input.

On-Screen Keyboards

Some individuals with physical disabilities cannot use a standard keyboard, but can use one or more switches or point with a mouse or headpointer (a device that lets users manipulate the mouse pointer on the screen through head motion). Groups of commands are displayed on the screen and the user employs a switch to choose a selected group, then a command within the group. Another form of this technique displays a picture of the keyboard allowing users to generate keystroke input on the screen by pointing to graphic images of the keys.

Keyboard Filters

Impaired physical abilities, such as erratic motion, tremors, or slow response, can sometimes be compensated by filtering out inappropriate keystrokes. The Windows Accessibility Options supports a wide range of keyboard filtering options. These are generally independent of the application with which users are interacting and therefore require no explicit support except for the standard system interfaces for keyboard input. However, users relying on these features can type slowly.

Compatibility with Screen Review Utilities

You can use the following techniques to ensure software compatibility with screen review utilities. The system allows your application to determine whether the system has been configured to provide support for a screen review utility (check the SM_SCREENREADER constant using the **GetSystemMetrics** function), allowing your software to enable or disable certain capabilities.

For more information about the SM_SCREENREADER constant, the **GetSystemMetrics** function, and other information about supporting screen review utilities, see the *Microsoft Win32 Programmer's Reference*.

Controls

Use standard Windows controls wherever possible. Most of these have already been implemented to support screen review utilities. Custom controls may not be usable by screen review utilities.

Always include a label for every control, even if you do not want the control's label to be visible. This applies regardless of whether you use standard controls or your own specialized controls (such as owner drawn controls or custom controls). If the control does not provide a label, you can create a label using a static text control.

Follow the normal layout conventions by placing the static text label before the control (above or to the left of the control). Also, set the keyboard TAB navigation order appropriately so that tabbing to a label navigates to the associated control it identifies instead of a label. To make certain that the label is recognized correctly, include a colon at the end of the label's text string. In cases where a label is not needed or would be visually distracting, provide the label, but do not make it visible. Although the label is not visible, it is accessible to a screen review utility.

Text labels are also effective for choices within a control. For example, you can enhance menus or lists that display colors or line widths by including some form of text representation, as shown in Figure 14.1.

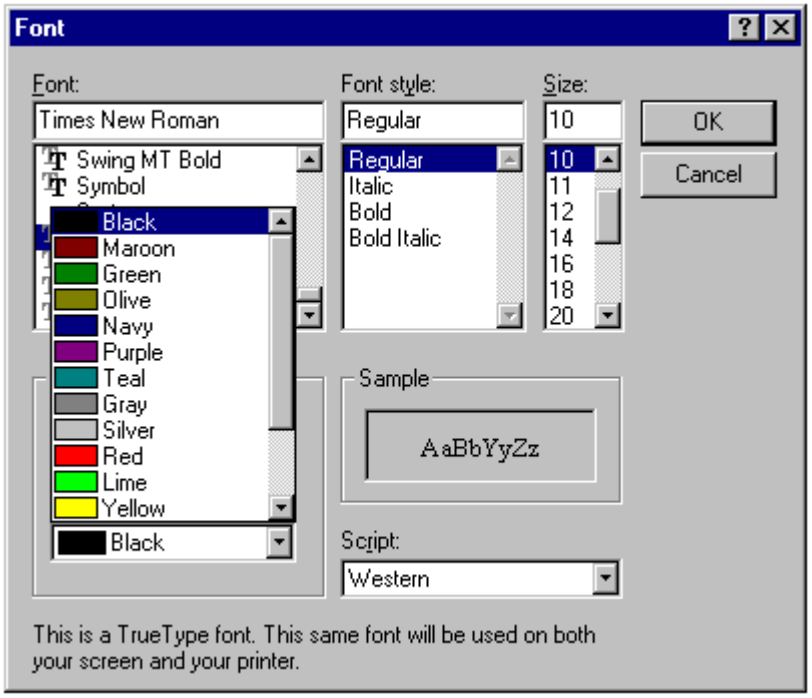


Figure 14.1 Using text to help identify choices

If providing a combined presentation is too difficult, offer users the choice between text and graphical representation, or choose one of them based on the system's screen review utility flag.

Text Output

Screen review utilities usually interpret text—including properties such as font, size, and face—that is displayed with standard system interfaces. However, text displayed as graphics (for example, bitmapped text) is not accessible to a screen review utility. To make it accessible, your application can create an invisible text label and associate the graphical representation of text with it. Screen review utilities can read standard text representations in a metafile, so you can also use metafiles instead of bitmap images for graphics information that includes text.

Graphics Output

Users with normal sight may be able to easily distinguish different elements of a graphic or pictorial information, such as a map or chart, even if they are drawn as a single image; however, a screen review utility must distinguish between different components. Ideally, use a metafile for graphics wherever possible.

When using bitmap images, consider separately drawing each component that requires identification. If performance is an issue, combine the component images in an off-screen bitmap using separate drawing operations and then display the bitmap on the screen with a single operation.

Alternatively, you can redraw each component with a null operation (NOP). This will not have an effect on the visible image, but allows a screen review utility to identify the region. You can also use this method to associate a text label with a graphic element.

Any of these methods can be omitted when the SM_SCREENREADER constant is not set. When drawing graphics, use standard Windows drawing functions. If you change an image directly—for example, clearing a bitmap by writing directly into its memory— a screen review utility does not recognize the content change and inappropriately describes it to users.

Icons and Windows

Accompany icons that represent objects with a text label (title) of the object's name. Use the system font and color for icon labels, and follow the system conventional for placement of the text relative to the icon. This allows a screen review utility to identify the object without special support. Similarly, make certain that all your windows have titles. Even if the title is not visible it is still available to access utilities. The more unique your window titles, the easier users can differentiate between them, especially when using a screen review utility. Using unique window class names is another way to provide for distinct window identification, but providing appropriate window titles is preferred.

The User's Point of Focus

Many accessibility aids must follow where the user is working. For example, a screen review utility conveys to users where the input focus is; a screen enlarger pans its viewport to ensure that users' focus is always kept on the visible portion of the screen. Most utilities give users the ability to manually move the viewport, but this becomes a laborious process, especially if it has to be repeated each time the input focus moves.

When the system handles the move of the input focus, through selecting a menu, navigating between controls in a dialog box, or activating a window, an accessibility utility tracks the changes. However, the utility may not detect when an application moves the input focus within its own window. Therefore, whenever possible, use standard system functions to place the input focus, such as the text insertion point. Even when you provide your own implementation of focus, you can use the system functions (such as the **SetCaretPos** function) to indicate focus location without making the standard input focus indicator visible.

For more information about the **SetCaretPos** function, see the *Microsoft Win32 Programmer's Reference*.

Timing and Navigational Interfaces

Some users read text or press keys very slowly, and do not respond to events as quickly as the average user. Avoid displaying critical feedback or presenting messages briefly and then automatically removing them because many users cannot read or respond to them. Similarly, limit your use of time-out based interfaces and always provide a way for users to configure them where you use them.

Also, avoid displaying or hiding information based on the movement of the pointer unless it is part of a standard system interface (for example, tooltips). Although such techniques can benefit some users, it may not be available for those using accessibility utilities. If you do provide such support, consider making these features optional so that users can turn them on or off when the SM_SCREENREADER constant is set.

Similarly, you should avoid using general navigation to trigger operations, because users of accessibility aids may need to navigate through all controls. For example, basic TAB keyboard navigation in a dialog box should not carry out the actions associated with a control, such as setting a check box or carrying out a command button. However, navigation can be used to faciliate further user interaction, such as validating user input or opening a drop-down control.

Color

Base the color properties of your interface elements on the system colors for window components, rather than hard coding specific colors. Remember to use appropriate foreground and background color combinations. If the foreground of an element is rendered with the button text color, use the button face color

as its background rather than the window background color. If the system does not provide standard color settings that can be applied to some elements, you can include your own interface that allows users to customize colors. In addition, you can provide graphical patterns as an optional substitute for colors as a way to distinguish information.

For more information about the use of color and how it is used for interface elements, see Chapter 13, "Visual Design."

The system also provides a global setting called High Contrast Mode that users can set through the Windows Accessibility Options. The setting provides a high contrast color setting between foreground and background visual elements.

Your application should check for this setting's status when it starts, and whenever it receives notification of system setting changes. When set, adjust your interface colors based on those set for the high contrast color scheme. In addition, whenever High Contrast Mode is set, hide any images that are drawn behind text (for example, watermarks or logos) to maintain the legibility of the information on the screen. You can also display monochrome versions of bitmaps and icons using the appropriate foreground color.

The accessibility High Contrast Mode status is available using the **HIGHCONTRAST** structure. For more information about this structure, see the *Microsoft Win32 Programmer's Reference*.

Scalability

Another important way to provide for visual accessibility is to allow for the scalability of screen elements. Sometimes, this simply means allowing users to change the font for the display of information. The system provides scalable fonts, controls, and functions that make it easy for the user to customize their interface. The system also provides a way for users to change the size and color of standard screen elements. You should use these same metrics for appropriately adjusting the size of other visual information you provide.

For more information about the system metrics for font and size, see Chapter 13, "Visual Design."

You can enlarge display information through scaling of your visual elements. The system already supports the scaling of standard Windows components. For your own custom elements, you can provide scaling by including a TrueType font or metafiles for your graphics images.

It may also useful to provide scaling features within your application. For example, many application provide a "Zoom" command that scales the presentation of the information displayed in a window, or other commands that make the presentation of information easier to read.

Keyboard and Mouse Interface

Providing a good keyboard interface is the most important step in accessibility because it affects users with a wide range of disabilities. For example, a keyboard interface may be the only option for users who are blind or use voice input utilities, and those who cannot use a mouse. The Windows Accessibility Options often compensate for users with disabilities related to keyboard interaction; however, it is more difficult to compensate for problems related to pointing device input.

You should follow the conventions for keyboard navigation techniques presented in this guide. For specialized interfaces within your software, model your keyboard interface on conventions that are familiar and appropriate for that context. Where they apply, use the standard control conventions as a guide for your defining interaction.

Make certain the user can navigate to all objects. Avoid relying only on navigational design that requires the user to understand the spatial relationship between objects. TAB and SHIFT+TAB can supplement arrow key navigation.

Providing a well-designed mouse interface is also important. Pointing devices may be the only means of interaction for some users. When designing the interface for pointing input, avoid making basic functions available only through multiple clicking, drag and drop manipulation, and keyboard-modified mouse actions. Such actions are best considered shortcut techniques for more advanced users. Make basic functions available through single click techniques.

Where possible, avoid making the implementation of basic functions dependent on a particular device. This is critical for supporting users with physical disabilities and users who may not wish to use or install a particular device.

Documentation, Packaging, and Support

While this guide focuses primarily on the design of the user interface, a design that provides for accessibility needs to take into consideration other aspects of a product. For example, consider the documentation needs of your users. For users who have difficulty reading or handling printed material, provide online documentation for your product.

If the documentation or installation instructions are not available online, you can provide documentation separately in alternative formats, such as ASCII text, large print, Braille, or audio tape format.

For more information about organizations that can help you produce and distribute such documentation, see the Bibliography.

When possible, choose a format and binding for your documentation that makes it accessible for users with disabilities. As in the interface, information in color should be a redundant form of communication. Bindings that allow a book to lie flat are usually better for users with limited dexterity.

Packaging is also important because many users with limited dexterity can have difficulty opening packages. Consider including an easy-opening overlap or tab that helps users remove shrink-wrapping.

Finally, although support is important for all users, it is difficult for users with hearing impairments to use standard support lines. Consider making these services available to customers using text telephones (also called "TT" or "TDD"). You can also provide support through public bulletin boards or other networking services.

Usability Testing

Just as it is important to test the general usability of your software, it is a good idea to test how well it provides for accessibility. There are a variety of ways of doing this. One way is to include users with disabilities in your prerelease or usability test activities. In addition, you can establish a working relationship with companies that provide accessibility aids. Finally, you can also try running your software in a fashion similar to that used by a person with disabilities. Try some of the following ideas for testing:

- Use the Windows Accessibility Options and set your display to a high contrast scheme, such as white text on a black background. Are there any portions of your software that become invisible or hard to use or recognize?

- Try using your software for a week without using a mouse. Are there operations that you cannot perform? Was anything especially awkward?
- Increase the size of the default system fonts. Does your software still look good? Does your software fonts appropriately adjust to match the new system font?

For more information about accessibility vendors or potential test sites, see the Bibliography.

Internationalization

To successfully compete in international markets, your software must easily accommodate differences in language, culture, and hardware. This section does not cover every aspect of preparing software for the international market, but it does summarize some of the key design issues.

The process of translating and adapting a software product for use in a different country is called *localization*. Like any part of the interface, include international considerations early in the design and development process. In addition to adapting screen information and documentation for international use, Help files, scenarios, templates, models, and sample files should all be a part of your localization planning.

For more information about the technical details for localizing your application, see the documentation included in the Microsoft Win32 Software Development Kit.

Language is not the only relevant factor when localizing an interface. Several countries can share a common language but have different conventions for expressing information. In addition, some countries can share a language but use a different keyboard convention.

A more subtle factor to consider when preparing software for international markets is cultural differences. For example, users in the U.S. recognize a rounded mail box with a flag on the side as an icon for a mail program, but this image may not be recognized by users in other countries.

It is helpful to create a supplemental document for your localization team that covers the terms and other translatable elements your software uses, and describes where they occur. Documenting changes between versions saves time in preparing new releases.

Text

A major aspect of localizing an interface involves translating the text used by the software in its title bars, menus and other controls, and messages. To make localization easier, store interface text as resources in the resource file rather than including it in the source code of the application. Remember to translate menu commands your application stores for its file types in the system registry.

Translation is a challenging task. Each foreign language has its own syntax and grammar. Following are some general guidelines to keep in mind for translation:

- Do not assume that a word always appears at the same location in a sentence, that word order is always the same, that sentences or words always have the same length, or that nouns, adjectives, and verbs always keep the same form.
- Avoid using vague words that can have several meanings in different contexts.
- Avoid colloquialisms, jargon, acronyms, and abbreviations.

- Use good grammar. Translation is a difficult enough task without a translator having to deal with poor grammar.
- Avoid dynamic, or run-time, concatenation of different strings to form new strings—for example, composing messages by combining frequently used strings. An exception is the construction of filenames and path.
- Avoid hard coding filenames in a binary file. Filenames may need to be translated.

Translation of interface text from English to other languages often increases the length of text by 30 percent or more. In some extreme cases, the character count can increase by more than 100 percent; for example, the word "move" becomes "verschieben" in German. Accordingly, if the amount of the space for displaying text is strictly limited, as in a status bar, restrict the length of the English interface text to approximately one half of the available space. In contexts that allow more flexibility, such as dialog boxes and property sheets, allow 30 percent for text expansion in the interface design. Message text in message boxes, however, should allow for text expansion of about 100 percent. Avoid having your software rely on the position of text in a control or window because translation may require movement of the text.

Expansion due to translation affects other aspects of your product. A localized version is likely to affect file sizes, potentially changing to the layout of your installation disks and setup software.

Translation is not always a one-to-one correspondence. A single word in English can have multiple translations in another language. Adjectives and articles sometimes change spelling according to the gender of the nouns they modify. Therefore, be careful when reusing a string in multiple places. Similarly, several English words may have only a single meaning in another language. This is particularly important when creating keywords for the Help index for your software.

Graphics

It is best to review the proposed graphics for international applicability early in your design cycle. Localizing graphics can be time consuming.

While graphics communicate more universally than text, graphical aspects of your software—especially icons and toolbar button images—may also need to be revised to address an international audience. Choose generic images and glyphs. Even if you can create custom designs for each language, having different images for different languages can confuse users who work with more than one language version.

Many symbols with a strong meaning in one culture do not have any meaning in another. For example, many symbols for U.S. holidays and seasons are not shared around the world. Importantly, some symbols can be offensive in some cultures (for example, the open palm commonly used at U.S. crosswalk signals is offensive in some countries).

Keyboards

International keyboards also differ from those used in the U.S. Avoid using punctuation character keys as shortcut keys because they are not always found on international keyboards or easily produced by the user. Remember too, that what seems like an effective shortcut because of its mnemonic association (for example, CTRL+B for Bold) can warrant a change to fit a particular language. Similarly, macros or other utilities that invoke menus or commands based on access keys are not likely to work in an international version, because the command names on which the access keys are based differ.

Keys do not always occupy the same positions on all international keyboards. Even when they do, the interpretation of the unmodified keystroke can be different. For example, on U.S. keyboards, SHIFT+8 results

in an asterisk character. However, on French keyboards, it generates the number 8. Similarly, avoid using CTRL+ALT combinations, because the system interprets this combination for some language versions as the ALTGR key, which generates some alphanumeric characters.

Your software can query standard system interfaces to determine keyboard configuration for a particular installation using the **SystemParametersInfo** function. For more information about the **SystemParametersInfo** function, see the *Microsoft Win32 Programmer's Reference*.

Character Sets

Some international countries require support for different character sets (sometimes called *code pages*). The system provides a standard interface for supporting multiple character sets and sort tables. Use these interfaces wherever possible for sorting and case conversion. In addition, consider the following guidelines:

- Do not assume that the character set is U.S. ANSI. Many ANSI character sets are available. For example, the Russian version of Windows 95 uses the Cyrillic ANSI character set which is different that the U.S. ANSI set.
- Use the system functions for supporting font selection (such as the common font dialog box).
- Always save the character set with font names in documents.

Formats

Different countries often use substantially different formats for dates, time, money, measurements, and telephone numbers. This collection of language-related user preferences are call a *locale*. Designing your software to accommodate international audiences requires supporting these different formats.

Windows provides a standard means for inquiring what the default format is and also allows the user to change those properties. Your software can allow the user to change formats, but restrict these changes to your application or document type, rather than affecting the system defaults. Table 14.1 lists the most common format categories.

Table 14.1 Formats for International Software

Category	Format considerations
Date	Order, separator, long or short formats, and leading zero
Time	Separator and cycle (12-hour versus 24-hour), leading zero
Physical quantity	Metric vs. English measurement system
Currency	Symbol and format (for example, trailing vs. preceding symbol)
Separators	List, decimal, and thousandths separator
Telephone numbers	Separators for area codes and exchanges
Calendar	Calendar used and starting day of the week
Addresses	Order and postal code format
Paper sizes	U.S. vs. European paper and envelope sizes

For more information about the functions that provide access to the current locale formats, see the *Microsoft Win32 Programmer's Reference*.

Layout

For layout of controls or other elements in a window, it is important to consider alignment in addition to expansion of text labels. In Hebrew and Arabic countries, information is written right to left. So when localizing for these countries, reverse your U.S. presentation.

Some languages include diacritical marks that distinguish particular characters. Fonts associated with these characters can require additional spacing.

In addition, do not place information or controls into the title bar area. This is where Windows places special user controls for configurations that support multiple languages.

References to Unsupported Features

Avoid confusing your international users by leaving in references to features that do not exist in their language version. Adapt the interface appropriately for features that do not apply. For example, some language versions may not include a grammar checker or support for bar codes on envelopes. Remove references to features such as menus, dialog boxes, and Help files from the installation program.

Network Computing

Windows provides an environment that allows the user to communicate and share information across the network. When designing your software, consider the special needs that working in such an environment requires.

Conceptually, the network is just an expansion of the user's local space. The interface for accessing objects from the network should not differ significantly from or be more complex than the user’s desktop.

Leverage System Support

Windows provides several provisions you can use when designing for network access:

- Use universal naming convention (UNC) paths to refer to objects stored in the file system. This convention provides transparent access to objects on the network.
- Use system-supported user identification that allows you to determine access without having to include your own password interface.
- Adjust window sizes and positions based on the local screen properties of the user.
- Avoid assuming the presence of a local hard disk. It is possible that some of your users work with diskless workstations.

Client-Server Applications

Users operating on a network may wish to run your application from a network server. For applications that store no state information, no special support is required. However, if your application stores state information, design your application with a server set of components and a client set of components. The server components include the main executable (.EXE) files, dynamic link libraries (DLLs), and any other files that need to be shared across the network. The client components consist of the components of the application that are specific to the user, including local registry information and local files that provide the user with access to the server components.

For information about installing the client and server components of your application, see Chapter 10, "Integrating with the System."

Shared Data Files

When storing a file in the shared space of the network, it should be readily accessible to all users, so design the file to be opened multiple times. The granularity of concurrent access depends on the file type; that is, for some files you may only support concurrent access by word, paragraph, section, page, and so on. Provide clear visual cues as to what information can be changed and what cannot. Where multiple access is not easily supported, provide users with the option to open a copy of the file, if the original is already open.

Records Processing

Record processing or transaction-based applications require somewhat different structuring than the typical productivity application. For example, rather than opening and saving discrete files, the interface for such applications focuses on accessing and presenting data as records through multiple views, forms, and reports. One of the distinguishing and most important design aspects of record-processing applications is the definition of how the data records are structured. This dictates what information can be stored and in what format.

However, you can apply much of the information in this guide to record-oriented applications. For example, the basic principles of design and methodology are just as applicable as they are for individual file-oriented applications. You can also apply the guide's conventions for input, navigation, and layout when designing forms and reports designs. Similarly, your can apply other secondary window conventions for data-entry design, including the following:

- Provide reasonable default values for fields.
- Use the appropriate controls. For example, use drop-down list boxes instead of long lists of push buttons.
- Design for logical and smooth user navigation. Order fields as the user needs to move through them. Auto-exit text boxes are often good for input of predefined data formats, such as time or currency inputs.
- Provide data validation as close to the site of data entry as possible. You can use input masks to restrict data to specific types or list box controls to restrict the range of input choices.

Telephony

Windows provides support for creating applications with telephone communications, or *telephony*, services. Those services include the Assisted Telephony services, for adding minimal, but useful telelphonic functionality to applications, and the full Telephony API, for implementing full telephonic applications.

For more information about creating application using the Microsoft Windows Telephony API (TAPI), see the documentation included in the Microsoft Win32 Software Developer's Kit.

Consider the following guidelines when developing telephony applications:

- Provide separate fields for users to enter country code, area code, and a local number. You may use auto-exit style navigation to facilitate the number entry. You can also use a drop-down list box to allow users to select a country code. (The TAPI **lineGetCountry** function will provide you will the list of available codes.)
- Provide access to the TAPI "Dialing Properties" property sheet window wherever a user enters a phone number. This window provides a consistent and easy interface for users.
- Use the modem configuration interfaces provided by the system. If the user has not installed a modem, run the Windows TAPI modem installation wizard.

Microsoft Exchange

Microsoft Exchange is the standard Windows interface for email, voice mail, FAX, and other communication media. Applications interact with Exchange by using the Messaging API (MAPI) and support services and components.

Microsoft Exchange allows you to create support for an information service. An information service is a utility that enables messaging applications to send and receive messages and files, store items in an information store, obtain user addressing information, or any combination of these functions.

Coexisting with Other Information Services

Microsoft Exchange is designed to simultaneously support different information services. Therefore when designing an information service, avoid:

- Initiating lengthy operations.
- Assuming exclusive use of key hardware resources, such as communications (COMM) ports and modems.
- Adding menu commands that might be incompatible with other services.

Adding Menu Items and Toolbar Buttons

Microsoft Exchange allows you to add menu items and toolbar buttons to the main viewer window. Follow the recommendations in this guide for defining menu and toolbar entries. In addition, where possible, define your menu items and toolbar entries (or their tooltips) in a way that allows the user to clearly associate the functionality with a specific information service.

Supporting Connections

When the user selects an information service that you support, provide the user with a dialog box to confirm the choice and allow the user access to configuration properties. Because simultaneous services run at the same time, clearly identify the service. Figure 14.2 provides an example of a typical connection window.

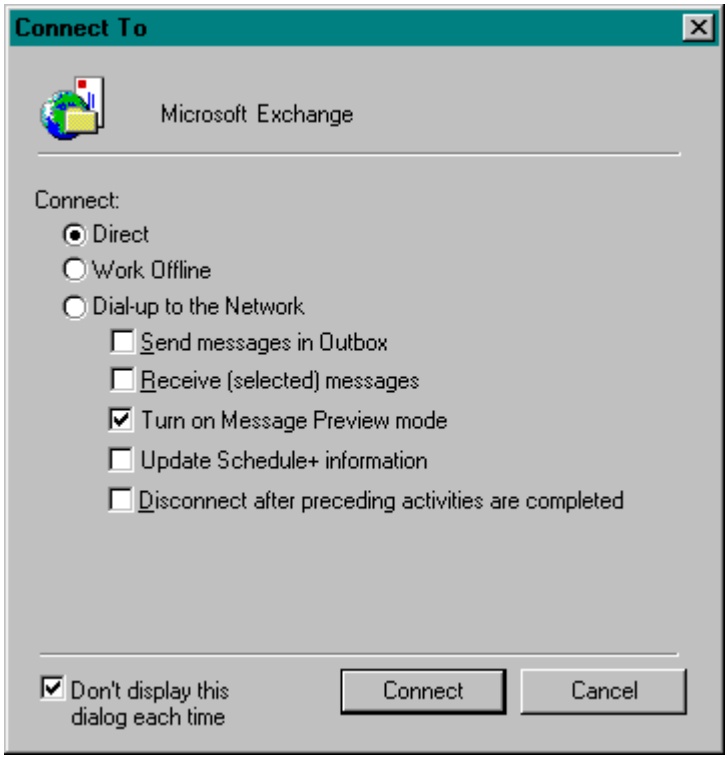


Figure 14.2 A connection dialog box

At the top of the window, display the icon and name of the service. You can include an option to not display the dialog box.

Installing Information Services

Microsoft Exchange includes a special wizard for installation of information services. You can support this wizard to allow the user to easily install your service.

The system also provides profiles and files that define which services are available to users when they log on. When the user installs your service, ask the user which profile they would like to include your service, such as their default profile.

APPENDIX A

Mouse Interface Summary

The tables in this appendix summarize the basic mouse interface, including selection and direct manipulation (drag and drop).

Table A. 1 Interaction Guidelines for Common, Unmodified Mouse Actions

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
Press.	Unselected object.	Clears the active selection.	Resets the anchor point to the object.	Selects the object.	Selects the object.
	Selected object.	None.	None.	None[1].	None.
	White space (background).	Clears the active selection.	Resets the anchor point to the button down location.	Initiates a region (marquee) selection.	Initiates a region (marquee) selection.
Click.	Unselected object.	Clears the active selection.	Resets the anchor point to the object.	Selects the object.	Selects the object and displays its pop-up menu.
	Selected object.	None[2].	None[2].	Selects the object[1].	Selects the object[1] and displays the selection's pop-up menu.
	White space (background).	Clears the active selection.	None.	None.	Displays the pop-up menu for the white space.
Drag.	Unselected object.	Clears the active selection.	Resets the anchor point to the object.	Selects the object and carries out the default transfer operation upon reaching destination.	Selects the object and displays the nondefault transfer pop-up menu upon reaching destination.

Table A. 1 Interaction Guidelines for Common, Unmodified Mouse Actions (*continued*)

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
Drag.	Selected object.	None.	None.	Carries out the default transfer operation on selection upon reaching destination.	Displays the nondefault transfer pop-up menu upon reaching destination.
	White space (background).	Clears the active selection.	None.	Selects everything logically included from anchor point to active end.	Selects everything logically included from anchor point to active end and displays pop-up menu for the resulting selection.
Double-click.	Unselected object.	Clears the active selection.	Resets the anchor point to the object.	Selects the object and carries out the default operation.	Selects the object and carries out the Properties command.
	Selected object.	None.	None.	Carries out the selection's default operation.	Carries out the selection's Properties command.
	White space (background).	Clears the active selection.	None.	Carries out the default operation for the white space[3].	Carries out the white space's Properties command[3].

[1]Alternatively, you can support subselection for this action. Subselection means to distinguish a specific object in a selection for some purpose. For example, in a selection of objects, subselecting an object may define that object as the reference point for alignment commands.

[2]Alternatively, you can support clearing the active selection and reset the anchor point to the object—if this better fits the context of the user's task.

[3] The white space (or background) is an access point for commands of the view, the container, or both. For example, white space can include view commands related to selection (Select All), magnification (Zoom), type of view (Outline), arrangement (Arrange By Date), display of specific view elements (Show Grid), general operation of the view (Refresh), and containment commands that insert objects (Paste).

Table A. 2 Interaction Guidelines for Using the SHIFT Key to Modify Mouse Actions

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
SHIFT+ Press.	Unselected object.	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object[2].	Extends the selection state from the anchor point to the object[3].
	Selected object.	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object[2].	Extends the selection state from the anchor point to the object[3].
	White space (background).	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object logically included at the button down point[2].	Extends the selection state from the anchor point to the object logically included at the button down point[3].
SHIFT+ Click.	Unselected object.	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object[2].	Extends the selection state from the anchor point to the object[2] and displays the pop-up menu for the resulting selection[3].
	Selected object.	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object[2].	Extends the selection state from the anchor point to the object[2] and displays the pop-up menu for the resulting selection[3].
	White space (background).	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object logically included at the button down point[2].	Extends the selection state from the anchor point to the object[2] logically included at the button down point and displays the pop-up menu for the resulting selection[3].

Table A. 2 Interaction Guidelines for Using the SHIFT Key to Modify Mouse Actions (*continued*)

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
SHIFT+ Drag.	Unselected object.	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object[2].	Extends the selection state from the anchor point to the object[2] and displays the pop-up menu for the resulting selection[3].
	Selected object.	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object[2].	Extends the selection state from the anchor point to the object[2] and displays the pop-up menu for the resulting selection[3].
	White space (background).	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object logically included at the button down point[2].	Extends the selection state from the anchor point to the object logically included at the button down point[2] and displays the pop-up menu for the resulting selection[3].
SHIFT+ Double-click.	Unselected object.	Clears the active selection[1].	Resets the anchor point to the object.	Extends the selection state from the anchor point to the object and carries out the default command on the resulting selection[2,3].	Extends the selection state from the anchor point to the object[2] and carries out the Properties command on the resulting selection[3].

Table A. 2 Interaction Guidelines for Using the SHIFT Key to Modify Mouse Actions (*continued*)

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
SHIFT+ Double-click.	Selected object.	None.	None.	Extends the selection state from the anchor point to the object and carries out the default command on the resulting selection[2,3].	Extends the selection state from the anchor point to the object[2] and carries out the Properties command on the resulting selection[3].
	White space (background).	Clears the active selection[1].	None.	Extends the selection state from the anchor point to the object logically included at the button down point[2] and carries out the default command on the resulting selection[3].	Extends the selection state from the anchor point to the object logically included at the button down point[2] and carries out the Properties command on the resulting selection[3].

[1] Only the active selection is cleared. The active selection is the selection made from the current anchor point. Other selections made by disjoint selection techniques are not affected, unless the new selection includes those selected elements.

[2] The resulting selection state is based on the selection state of the object at the anchor point. If that object is selected, all the objects included in the range are selected. If the object is not selected, all the objects included in the range are also not selected.

[3] If the effect of extending the selection results unselects the object or a range of objects, the operation applies also to the remaining selected objects.

Table A. 3 Interaction Guidelines for Using the CTRL Key to Modify Mouse Actions

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
CTRL+ Press.	Unselected object.	None.	Resets the anchor point to the object.	Selects the object[1].	Selects the object[1].
	Selected object.	None.	Resets the anchor point to the object.	None.	None.
	White space (background).	None.	Resets the anchor point to the button down location.	Initiates a disjoint region selection.	Initiates a disjoint region selection.
CTRL+ Click.	Unselected object.	None.	Resets the anchor point to the object.	Selects the object[1].	Selects the object[1] and displays the pop-up menu for the entire selection.
	Selected object.	None.	Resets the anchor point to the object.	Unselects the object[1].	Unselects the object[1]and displays the pop-up menu for the remaining selection.
	White space (background).	None.	None.	None.	Displays the pop-up menu for the existing selection.

Table A. 3 Interaction Guidelines for Using the CTRL Key to Modify Mouse Actions (*continued*)

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
CTRL+ Drag.	Unselected object.	None.	Resets the anchor point to the object.	Selects the object[1] and copies the entire selection[2].	Selects the object[1] and displays the pop-up menu for the selection [3].
	Selected object.	None.	Resets the anchor point to the object.	Copies the entire selection to the destination defined at the button up location[2].	Copies the entire selection to the destination defined at the button up location[3].
	White space (background).	None.	None.	Toggles the selection state of objects logically included by region selection[4].	Toggles the selection state of objects logically included by region selection[4] and displays the pop-up menu for the resulting selection[5].

Table A. 3 Interaction Guidelines for Using the CTRL Key to Modify Mouse Actions (*continued*)

Action	Target	Effect on selection state	Effect on anchor point location	Resulting operation using button 1	Resulting operation using button 2
CTRL+ Double-click.	Unselected object.	None.	Resets the anchor point to the object.	Toggles the selection state of the object (selects) and carries out the default command on the selection set.	Toggles the selection state of the object (selects) and carries out the Properties command on the selection set.
	Selected object.	None.	Resets the anchor point to the object.	Toggles the selection state of the object (selects) and carries out the default command on the selection set[5].	Toggles the selection state of the object (selects) and carries out the Properties command on the selection set[5].
	White space (background).	None.	None.	Carries out the default command on the existing selection.	Carries out the Properties command on the white space[6].

[1] The CTRL key toggles the selection state of an object; this table entry shows the result.

[2] If the user releases the CTRL key before releasing the mouse button, the operation reverts to the default transfer operation (as determined by the destination).

[3] If the user releases the CTRL key before releasing the mouse button, the operation reverts to displaying the nondefault transfer (drag and drop) pop-up menu.

[4] The range of objects included are all toggled to the same selection state, which is based on the first object included by the bounding region (marquee).

[5] If the effect of toggling cancels the selection of the object, the operation applies to the remaining selected objects.

[6] The white space (background) is an access point to the commands of the view, the container, or both.

APPENDIX B

Keyboard Interface Summary

This appendix summarizes the common keyboard operations, shortcut keys, and access key assignments.

Table B.1 displays a summary of the keys commonly used for navigation.

Table B.1 Common Navigation Keys

Key	Cursor movement	CTRL+cursor movement
LEFT ARROW	Left one unit.	Left one proportionally larger unit.
RIGHT ARROW	Right one unit.	Right one proportionally larger unit.
UP ARROW	Up one unit or line.	Up one proportionally larger unit.
DOWN ARROW	Down one unit or line.	Down one proportionally larger unit.
HOME	To the beginning of the line.	To the beginning of the data (topmost position).
END	To the end of the line.	To the end of the data (bottommost position).
PAGE UP	Up one screen (previous screen, same position).	Left one screen (or previous unit, if left is not meaningful).
PAGE DOWN	Down one screen (next screen, same position).	Right one screen (or next unit, if right is not meaningful).
TAB[1]	Next field.	To next tab position (in property sheets, next page).

[1] Using the SHIFT key with the TAB key navigates in the reverse direction.

Table B.2 lists the common shortcut keys. Avoid assigning these keys to functions other than those listed.

Table B.2 Common Shortcut Keys

Key	Meaning
TRL+C	opy
TRL+O	pen
TRL+P	rint
TRL+S	ave
TRL+V	aste
TRL+X	ut
TRL+Z	ndo
1	isplay contextual Help window
HIFT+F1	ctivate contextual Help mode (What's This?)
HIFT+F10	isplay pop-up menu
PACEBAR[1]	elect (same as mouse button 1 click)
SC	ancel
LT	ctivate or inactivate menu bar mode
LT+TAB[2]	isplay next primary window (or application)
LT+ESC[2]	isplay next window
LT+SPACEBAR	isplay pop-up menu for the window
LT+HYPHEN	isplay pop-up menu for the active child window
LT+ENTER	isplay property sheet for current selection
LT+F4	lose active window
LT+F6[2]	witch to next window within application (between modeless secondary windows and their primary window)
LT+PRINT SCREEN	apture screen to Clipboard
TRL+ESC	ccess Start button in taskbar
TRL+ALT+DEL	isplay system's Close Program dialog box

[1] If the context (for example, a text box) uses the SPACEBAR for entering a space character, you can use CTRL+SPACEBAR. If that is also defined by the context, define your own key.

[2] Using the SHIFT key with this key combination navigates in the reverse direction.

Table B.3 lists shortcut key assignments for keyboards supporting the new Windows keys. The Left Windows key and Right Windows key are handled the same. Windows key assignments are reserved for Windows shell functions.

Table B.3 Windows Keys

Key	Meaning
PPLICATION key	Display pop-up menu for the selected object.
/INDOWS key	Display Start button menu.
/INDOWS+F1	Display Help Topics browser dialog box for the main Windows Help file.
/INDOWS+TAB	Activate next taskbar button.
/INDOWS+E	Explore My Computer.
/INDOWS+F	Find Document.
/INDOWS+CTRL+F	Find Computer.
/INDOWS+M	Minimize All.
HIFT+WINDOWS+M	Undo Minimize All.
/INDOWS+R	Display Run dialog box.
/INDOWS+BREAK	System function.

Table B.4 lists the key combinations and sequences the system uses to support accessibility. Support for these options is set by users with the Windows Accessibility Options.

Table B.4 Accessibility Keys

Key	Meaning
EFT ALT+LEFT SHIFT+PRINT SCREEN	Toggle High Contrast mode
EFT ALT+LEFT SHIFT+NUM LOCK	Toggle MouseKeys
HIFT (pressed five consecutive times)	Toggle StickyKeys
IGHT SHIFT (held eight or more seconds)	Toggle FilterKeys (SlowKeys, RepeatKeys, and BounceKeys)
UM LOCK (held five or more seconds)	Toggle ToggleKeys

Table B.5 lists the recommended access key assignments for common commands. While the context of a command may affect specific assignments, you should use the same access keys when you use these commands in your menus and command buttons.

Table B.5 Access Key Assignments

About	Insert Object	Quick View
Always on Top	Link Here	Redo
Apply	Maximize	Repeat
Back	Minimize	Restore
Browse	Move	Resume
Close	Move Here	Retry
Copy	New	Run
Copy Here	Next	Save
Create Shortcut	No	Save As
Create Shortcut Here	Open	Select All
Cut	Open With	Send To
Delete	Paste	Show
Edit	Paste Link	Size
Exit	Paste Shortcut	Split
Explore	Page Setup	Stop
File	Paste Special	Undo
Find	Pause	View
Help	Play	What's This?
Help Topics	Print	Window
Hide	Print Here	Yes
Insert	Properties	

Avoid assigning access keys to OK and Cancel when the ENTER key and ESC key, respectively, are assigned to them by default.

APPENDIX C

Guidelines Summary

The following checklist summarizes the guidelines covered in this guide. You can use this guideline summary to assist you in your planning, design, and development process.

Remember, the objective of the recommendations and suggestions in this guide is to benefit your users, not to enforce a rigid set of rules. Consistency in design makes it easier for a user to transfer skills from one task to another. When you need to diverge from or extend these guidelines, follow the principles and spirit of this guide.

General Design

- ☐ Supports user initiation of actions
- ☐ Supports user customization of the interface
- ☐ Supports an interactive and modeless environment
- ☐ Supports direct manipulation interfaces
- ☐ Uses familiar, appropriate metaphors
- ☐ Is internally consistent; similiar actions have a similar interface
- ☐ Makes actions reversible where possible; where not possible, requests confirmation
- ☐ Makes error recovery easy
- ☐ Eliminates possibilities for user errors, where possible
- ☐ Uses visual cues to indicate user interaction
- ☐ Provides prompt feedback
- ☐ Provides feedback that is appropriate to the task
- ☐ Makes appropriate use of progressive disclosure

Design Process

- ☐ Employs a balanced team
- ☐ Uses an iterative design cycle
- ☐ Incorporates usability assessment as a part of the process
- ☐ Designs for user limitations

Input and Interaction

- ☐ Follows basic mouse interaction guidelines
- ☐ Uses appropriate modifier keys for adjusting or adding elements to a selection
- ☐ Uses appropriate visual feedback, such as highlighting or handles, to indicate selected objects
- ☐ Supports default and nondefault drag and drop
- ☐ Supports standard transfer commands, where appropriate
- ☐ Provides keyboard interface for all basic operations
- ☐ Follows keyboard guidelines for navigation, shortcut keys, and access keys
- ☐ Keeps foreground activity as modeless as possible
- ☐ Indicates use of modes visually
- ☐ Provides access to common, basic operations through single click interaction
- ☐ Provides shortcut methods (such as double-clicking) to common or frequently used operations for experienced users

Windows

- ☐ Provides title text for all windows and follows guidelines for defining correct title bar text
- ☐ Supports single window instance model: brings the existing window to the top of the Z order when the user attempts to reopen a view or window that is already open
- ☐ Uses common dialog boxes, where applicable
- ☐ Saves and restores the window state
- ☐ Adjusts window size and position to the appropriate screen size
- ☐ Uses modeless secondary windows, wherever possible
- ☐ Limits the use of application modal secondary windows
- ☐ Avoids system modal secondary windows, except in the case of possible loss of data
- ☐ Automatically supplies a proposed name upon the creation of a new object
- ☐ Uses the appropriate message symbol in message boxes
- ☐ Provides a brief but clear statement of problem and possible remedies in message boxes

- ☐ Organizes properties into property sheets, using property pages for peer properties and list controls for hierarchical navigation
- ☐ Places command buttons that apply to the page inside a tabbed page (for example, a property sheet) outside of a page when the user applies by window (as a set)
- ☐ Follows single document window interface (SDI) or multiple document interface (MDI or MDI alternatives) conventions

Controls

- ☐ Use system-supplied controls wherever possible
- ☐ Provide a pop-up menu for the title bar icon
- ☐ Provide a pop-up menu for the window
- ☐ Avoid multiple level hierarchical interfaces (menus, secondary windows) for frequently used access operations
- ☐ Use an ellipsis for commands that require a dialog box for additional input or parameters
- ☐ Use the menu (triangular arrow) glyph to indicate when a control can display more information (cascading menus, drop-down control arrows, scroll bar arrows)
- ☐ Provide pop-up menus for selections and other user identifiable objects
- ☐ Support the display of pop-up menus using mouse button 2, SHIFT+F10, and action handles
- ☐ Display pop-up menus upon the release of the mouse button
- ☐ Follow guidelines in the order of the commands on pop-up menus
- ☐ Limit commands on pop-up menus to those that apply to the selection and its immediate context
- ☐ Make toolbars user configurable (display, position, content)
- ☐ Define custom controls to be visually and operationally consistent with standard system controls

Integrating with the System

- ☐ Makes full and correct use of the registry, including registration of file extensions, file types, and icons
- ☐ Avoids use of Autoexec.bat, Config.sys, or Windows system .Ini files
- ☐ Supports print and print-to interface, for file types that are printable
- ☐ Provides and registers icons in 32-by -32, 16-by -16, and 48-by -48 pixel sizes for application, and document and data file types
- ☐ Adds appropriate property pages for supported types
- ☐ Supports long filenames and universal naming convention (UNC) paths, where files are used
- ☐ Displays filenames correctly

- ☐ Uses the taskbar to provide a user with notification and status information when a window is not active
- ☐ Supports appropriate behavior for creating and integrating Scrap objects
- ☐ Follows guidelines for installation
- ☐ Provides an uninstall program
- ☐ Provides appropriate support for network installation
- ☐ Supports all OLE user interface guidelines, including transfer interfaces (drag and drop and nondefault drag and drop), pop-up menus and property sheets for OLE embedded and linked objects

User Assistance

- ☐ Provides context-sensitive Help information for elements (including controls)
- ☐ Provides task help topics for basic procedures
- ☐ Provides tooltips for all unlabeled controls, particularly in toolbars
- ☐ Follows guidelines for messages, status bar information, contextual Help, task Help, and on-line Reference Help

Visual Design

- ☐ Uses color only as an enhancing, secondary form of information
- ☐ Uses a limited set of colors
- ☐ Uses system metrics for all display elements (such as color settings and fonts)
- ☐ Uses standard border styles
- ☐ Uses appropriate appearance for visual states of controls
- ☐ Supports dimensionality using light source from the upper left
- ☐ Supports guidelines for layout and font use
- ☐ Uses correct capitalization for control labels

Sound

- ☐ Uses audio only for secondary cues (applicable only where audio is not the primary form of information, for example, music)
- ☐ Supports system interface for adjusting sound volume
- ☐ Supports and provides appropriate visual output for system ShowSounds setting

Accessibility

- ☐ Clearly labels all controls, icons, windows, and other screen elements (even if not visible) to make them available to screen review utilities
- ☐ Indicates keyboard focus
- ☐ Uses standard functions for displaying text
- ☐ Makes components of graphic images that must be separately discernible by using metafiles, drawing each component separately, or by redrawing components with null operation (NOP) when the user has installed a screen review utility
- ☐ Avoids time-out interaction or makes timing interaction user configurable
- ☐ Avoids triggering actions on user navigation in the interface
- ☐ Supports scaling or magnification views where possible and applicable
- ☐ Supports system accessibility settings (such as High Contrast Mode) and appropriately adjusts the user interface elements
- ☐ Tests for compatibility with common accessibility aids
- ☐ Includes people with disabilities in testing process
- ☐ Provides documentation in non-printed formats, such as on-line
- ☐ Provides telephone support to users using text telephones (TT/TDD)

International Users

- ☐ Provides sufficient space for character expansion for localization

☐ Avoids jargon and culturally dependent words or ideas

- ☐ Avoids using punctuation keys in shortcut key combinations
- ☐ Supports displaying information based on local formats
- ☐ Uses layout conventions appropriate to reading conventions
- ☐ Adjusts references to unsupported features

Network Users

- ☐ Supports system naming and identification conventions
- ☐ Supports shared access for application and data files

A P P E N D I X D

Supporting Windows 95 and Windows NT Version 3.51

To be supplied.

APPENDIX E

Localization Word Lists

This appendix will contain more than 25 translations of the following word list. Although the individual words in the list are subject to change, the intent of the list will remain the same: to provide a comprehensive set of words and phrases that either appear in the Windows 95 user interface or are used in describing key concepts of the operating system. Note that bold indicates command names that appear on buttons and menus.

1 **About**
2 access key
3 accessibility
4 action handle
5 active
6 active end
7 active object
8 active window
9 adornment
10 **Always on Top**
11 anchor point
12 **Apply**
13 auto-exit
14 auto-repeat
15 automatic link
16 automatic scrolling (autoscroll)
17 **Back**
18 barrel button (pen)
19 barrel-tap
20 boxed edit (control)
21 **Browse**
22 **Cancel**
23 cascading menu
24 check box
25 check mark
26 child window
27 choose
28 click
29 Clipboard
30 **Close**
31 Close button
32 collapse (outline)
33 column heading (control)
34 combo box
35 command button

36 container
37 context-sensitive help
38 contextual
39 control
40 **Copy**
41 **Copy Here**
42 **Create Shortcut**
43 **Create Shortcut Here**
44 **Cut**
45 default
46 default button
47 **Delete**
48 desktop
49 destination
50 dialog box
51 disability
52 disjoint selection
53 dock
54 document
55 double-click
56 double-tap
57 drag
58 drag and drop
59 drop-down combo box
60 drop-down list box

61 drop-down menu
62 **Edit**
63 **Edit menu**
64 ellipsis
65 embedded object
66 **Exit**
67 expand (an outline)
68 **Explore**
69 extended selection
70 extended selection list box
71 file
72 **File menu**
73 **Find**
74 **Find Next**
75 **Find What**
76 folder
77 font
78 font size
79 font style
80 function key
81 gesture
82 glyph
83 group box
84 handle
85 **Help**
86 **Help menu**
87 **Hide**
88 hierarchical selection
89 hold
90 hot spot
91 hot zone
92 icon
93 inactive
94 inactive window
95 ink

96 ink edit
97 input focus
98 **Insert menu**
99 **Insert Object**
100 insertion point
101 italic
102 label
103 landscape
104 lasso-tap
105 lens (control)
106 link (n.)
107 link (v.)
108 **Link Here**
109 list box
110 list view (control)
111 manual link
112 **Maximize**
113 maximize button
114 menu
115 menu bar
116 menu button
117 menu item
118 menu title
119 message box
120 **Minimize**
121 minimize button
122 mixed-value
123 modal
124 mode
125 modeless
126 modifier key
127 mouse
128 **Move**
129 **Move Here**
130 multiple document interface (MDI)

131 multiple selection list box
132 My Computer (icon)
133 Network Neighborhood (icon)
134 **New**
135 **Next**
136 object
137 **OK**
138 OLE
139 OLE drag and drop
140 OLE embedded object
141 OLE linked object
142 OLE nondefault drag and drop
143 **Open**
144 **Open With**
145 option button
146 option set
147 package
148 **Page Setup**
149 palette window
150 pane
151 parent window
152 password
153 **Paste**
154 **Paste Link**
155 **Paste Shortcut**
156 **Paste Special**
157 path
158 **Pause**
159 pen
160 **Play**
161 Plug and Play
162 point
163 pointer
164 pop-up menu
165 pop-up window

166 portrait
167 press (a key)
168 press (and hold a mouse button)
169 primary container
170 primary window
171 **Print**
172 printer
173 progress indicator (control)
174 project
175 **Properties**
176 property inspector
177 property page
178 property sheet
179 property sheet control
180 **Quick View**
181 read-only
182 recognition
183 Recycle Bin (icon)
184 **Redo**
185 region selection
186 registry
187 **Repeat**
188 **Replace**
189 **Restore**
190 Restore button
191 **Resume**
192 **Retry**
193 rich-text box
194 **Run**
195 **Save**
196 **Save As**
197 scroll
198 scroll arrow
199 scroll bar
200 scroll box
201 secondary window
202 select
203 **Select All**
204 selection
205 selection handle
206 **Send To**
207 separator
208 **Settings**
209 **Setup**
210 shortcut

211 shortcut button
212 shortcut icon
213 shortcut key
214 shortcut key control
215 **Show**
216 **Shut Down**
217 single selection list box
218 **Size**
219 size grip
220 slider
221 spin box
222 **Split**
223 split bar
224 split box
225 Start button
226 StartUp folder
227 status bar
228 **Stop**
229 tab control
230 tap
231 taskbar
232 task-oriented Help
233 template
234 text box
235 title bar
236 title text
237 toggle key
238 toolbar
239 tooltip
240 tree view control
241 type (n.)
242 type (v.)
243 unavailable
244 **Undo**
245 **Uninstall**
246 **View menu**
247 visual editing
248 well control
249 **What's This?**
250 window
251 **Window menu**
252 Windows Explorer
253 wizard
254 workbook
255 workgroup

256 workspace
257 **Yes**

Bibliography

General Design

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison-Wesley Pub. Co., 1975.

Baecker, Ronald M., and Buxton, William A. S. *Readings in Human-Computer Interaction: A Multidisciplinary Approach*. Los Altos, Calif.: M. Kaufmann, 1987.

Heckel, Paul. *The Elements of Friendly Software Design.* New Ed., San Francisco: SYBEX, 1991.

Lakoff, George, and Johnson, Mark. *Metaphors We Live By*. Chicago: University of Chicago Press, 1980.

Laurel, Brenda, Ed. *The Art of Human-Computer Interface Design*. Reading, Mass.: Addison-Wesley Pub. Co., 1990.

Norman, Donald A. *The Design of Everyday Things*. New York: Basic Books, 1990.

Norman, Donald A., and Draper, Stephen, W., Eds. *User Centered System Design*: *New Perspectives on Human-Computer Interaction*. Hillsdale, N.J.: L. Erlbaum Associates, 1986.

Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, Mass.: Addison-Wesley, 1992.

Tognazzini, Bruce. *Tog on Interface*. Reading, Mass.: Addison-Wesley, 1992.

Graphic Information Design

Blair, Preston. *Cartoon Animation*. How to Draw and Paint Series. Tustin, Calif.: Walter Foster Pub., 1989.

Dreyfuss, Henry. *Symbol Sourcebook: An Authoritative Guide to International Graphic Symbols*. New York: Van Nostrand Reinhold Co., 1984.

Thomas, Frank., and Johnston, Ollie. *Disney Animation: The Illusion of Life*. New York: Abbeville Press, 1984.

Tufte, Edward R. *Envisioning Information*. Cheshire, Conn.: Graphics Press, 1990.

Tufte, Edward R. *The Visual Display of Quantitative Information*. Cheshire, Conn.: Graphics Press, 1983.

Usability

Dumas, Joseph S., and Redish, Janice C., *A Practical Guide to Usability Testing.* Norwood, N.J.: Ablex Pub. Corp., 1993.

Nielsen, Jakob. *Usability Engineering*. Boston: Academic Press, 1993.

Rubin, Jeffrey. *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*. New York: Wiley, 1994.

Whiteside, John, Bennett, John, and Holtzblatt, Karen. "Usability Engineering: Our Experience and Evolution." In *Handbook of Human-Computer Interaction*, Martin. Helander (Ed.), Elsevier Science Pub. Co., Amsterdam, 1988. (pp. 791 - 817)

Wiklund, Michael E., Ed. *Usability in Practice: How Companies Develop User-Friendly Products*. Boston: AP Professional, 1994.

Object-Oriented Design

Booch, Grady. *Object-Oriented Analysis and Design with Applications*. Redwood City, Calif.: Benjamin/Cummings Pub. Co., 1994.

Peterson, Gerald E., Ed. *Tutorial: Object-Oriented Computing: Volume 2: Implementations*. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, N.J.: Prentice Hall, 1991.

Accessibility

For a list of accessibility aids available for Microsoft Windows, accessibility software vendors, or potential test sites, contact:

Microsoft Sales Information Center
One Microsoft Way
Redmond, WA 98052-6399

(800) 426-9400 (voice)
(800) 892-5234 (text telephone)
(206) 936-7329 (fax)

An assistive technology program in your area can provide referrals to programs and services available to you. To locate the assistive technology program nearest to your location, contact:

National Information System
Center for Development Disabilities
University of South Carolina
Benson Building
Columbia, SC 29208

(803) 777-4435 (voice or text telephone)
(803) 777-6058 (fax)

The Trace Research and Development Center publishes references and materials on accessibility, including:

Berliss, Jane R., Ed. *Trace Resource Book: Assistive Technologies for Communication, Control and Computer Access*. Madison, Wis.: Trace Research and Development Center, 1994.

Vanderheiden, Gregg C., and Vanderheiden, Katherine R. *Accessible Design of Consumer Products: Guidelines for the Design of Consumer Products to Increase Their Accessibility to People with Disabilities or Who Are Aging.* Madison, Wis.: Trace Research and Development Center, 1991.

For information on these books and other resources available from the Trace Research and Development Center, contact them at:

Trace Research and Development Center
University of Wisconsin - Madison
S-151 Waisman Center
1500 Highland Avenue
Madison, WI 5705-2280

(608) 263-2309 voice
(608) 263-5408 text telephone
(608) 263-8848 fax

Organizations

The following organizations publish journals and sponsor conferences on topics related to user interface design.

SIGCHI (Special Interest Group in Computer Human Interaction)
Association for Computing Machinery
1515 Broadway
New York, NY 10036-5701
212-869-7440

SIGGRAPH (Special Interest Group on Graphics)
Association for Computing Machinery

1515 Broadway
New York, NY 10036-5701
212-869-7440

Human Factors and Ergonomics Society
P.O. Box 1369
Santa Monica, CA 90406-1369
310-394-1811

Glossary

accelerator key

See shortcut key.

access key

The key that corresponds to an underlined letter on a menu or button (also referred to as a mnemonic or mnemonic access key).

accessibility

A software design that makes it usable and accessible to the widest range of users, including users with disabilities.

action handle

A special handle that provides access to a selected object's operations, typically by displaying a pop-up menu, through drag and drop, or both. Pen-enabled interfaces or controls include action handles more frequently than mouse or keyboard.

active

The state when an object is the focus of user input.

active end

The ending point for a selected range of objects. It is usually established at the object logically nearest the hot spot of the pointer when a user releases a mouse button or lifts the tip of a pen from the screen. *Compare* anchor point.

active window

The window in which a user is currently working or directing input. An active window is typically at the top of the Z order and is distinguished by the color of its title bar. *Compare* inactive window.

adornment

A control that is attached to the edge of a pane or window, such as a toolbar or ruler.

anchor point

The starting point for a selected range of objects. An anchor point is usually established at the object logically nearest the hot spot of the pointer when a user presses a mouse button or touches the tip of a pen to the screen. *Compare* active end.

anti-aliasing

A graphic design technique that involves adding colored pixels to smooth the jagged edges of a graphic.

apply

To commit a set of changes or pending transactions made in a secondary window, typically without closing that window.

auto-exit

A text box in which the input focus automatically moves to the next control as soon as a user types the last character.

auto-joining

The movement of text to fill a remaining gap after a user deletes other text.

automatic scrolling

A technique where a display area automatically scrolls without direct interaction with a scroll bar.

auto-repeat

An event or interaction that is automatically repeated. Auto-repeat events usually occur when a user holds down a keyboard key or presses and holds a special control (for example, scroll bar buttons).

barrel-tap

A pen action that involves holding down the barrel button of a pen while tapping. It is the equivalent of clicking mouse button 2.

box edit

A standard Microsoft Windows pen interface control that provides a discrete area for entering each character. A user can also edit text within the control.

cancel

To halt an operation or process and return to the state before it was invoked. *Compare* stop.

caret

See insertion point.

cascading menu

A menu that is a submenu of a menu item (also referred to as a hierarchical menu, child menu, or submenu).

check box

A standard Windows control that displays a setting, either checked (set) or unchecked (not set). *Compare* option button.

child menu

See cascading menu.

child window

A document window used within an MDI window. *See also* multiple document interface.

chord

To press more than one mouse button at the same time.

click

(v.) To position the pointer over an object and then press and release a mouse button. (n.) The act of clicking. *See also* press.

Clipboard

The area of storage for objects, data, or their references after a user carries out a Cut or Copy command.

close

To remove a window.

code page

A collection of characters that make up a character set.

collection

A set of objects that share some common aspect.

column heading

A standard Windows control that displays information in a multicolumn list.

combo box

A standard Windows control that combines a text box and interdependent list box. *Compare* drop-down combo box.

command button

A standard Windows control that initiates a command or sets an option (also referred to as a push button).

composite

A set or group of objects whose aggregation is recognized as an object itself (for example, characters in a paragraph, a named range of cells in a spreadsheet, or a grouped set of drawing objects).

constraint

A relationship between a set of objects, such that making a change to one object affects another object in the set.

container

An object that holds other objects.

context menu

See pop-up menu.

context-sensitive Help

Information about an object and its current condition. It answers the questions “What is this” and “Why would I want to use it?” *Compare* reference Help and task-oriented Help.

contextual

Specific to the conditions in which something exists or occurs.

contiguous selection

A selection that consists of a set of objects that are logically sequential or adjacent to each other (also referred to as range selection). *Compare* disjoint selection.

control

An object that enables user interaction or input, often to initiate an action, display information, or set values.

Control menu

The menu, also referred to as the System menu, was displayed on the left end of a title bar in Windows 3.1. A pop-up menu of a window replaces the Control menu.

cursor

A generic term for the visible indication of where a user's interaction will occur. *See also* input focus, insertion point, and pointer.

data-centered design

A design in which users interact with their data directly without having to first start an appropriate editor or application.

data link

A link that propagates a value between two objects or locations.

default

An operation or value that the system or application assumes, unless a user makes an explicit choice.

default button

The command button that is invoked when a user presses the ENTER key. A default button typically appears in a secondary window.

delete

To remove an object or value.

desktop

The visual work area that fills the display. The desktop is also a container and can be used as a convenient location to place objects stored in the file system.

dialog base unit

A device-independent measure to use for layout. One horizontal unit is equal to one-fourth of the average character width for the current system font. One vertical unit is equal to one-eighth of an average character height for the current system font.

dialog box

A secondary window that gathers additional information from a user. *Compare* message box, palette window, and property sheet.

dimmed

See unavailable.

disability

A skill level that is near the lower range for an average person.

disabled

See unavailable.

disjoint selection

A selection that consists of a set of objects that are not logically sequential or physically adjacent to each other. *Compare* contiguous selection. *See also* extended selection.

dock

To manipulate an interface element, such as a toolbar, such that it aligns itself with the edge of another interface element, typically a window or pane.

document

A common unit of data (typically a file) used in user tasks and exchanged between users.

document window

A window that provides a primary view of a document (typically its content).

double-click

(v.) To press and release a mouse button twice in rapid succession. (n.) The act of double-clicking.

double-tap

(v.) To press and lift the pen tip twice in rapid succession. It is typically interpreted as the double-click of the mouse.
(n.) The act of double-tapping.

drag

To press and hold a mouse button (or press the pen tip) while moving the mouse (or pen).

drag and drop

A technique for moving, copying, or linking an object by dragging. The source and destination negotiate the interpretation of the operation. *Compare* nondefault drag and drop.

drop-down combo box

A standard Windows control that combines the characteristics of a text box with a drop-down list box. *Compare* combo box.

drop-down list box

A standard Windows control that displays a current setting, but can be opened to display a list of choices.

drop-down menu

A menu that is displayed from a menu bar. *See also* menu and pop-up menu.

edit field

See text box.

Edit menu

A common drop-down menu which includes general purpose commands, such as Cut, Copy, and Paste for editing objects displayed within a window.

ellipsis

The "..." suffix added to a menu item or button label to indicate that the command requires additional information to be completed. When a user chooses the command, a dialog box is usually displayed for user input of this additional information.

embedded object

See OLE embedded object.

event

An action or occurrence to which an application can respond. Examples of events are clicks, key presses, and mouse movements.

explicit selection

A selection that a user intentionally performs with an input device. *See also* implicit selection.

extended selection

A selection technique that is optimized for selection of a single object or single range using contiguous selection techniques (that is, canceling any existing selection when a new selection is made). However, it also supports modifying an existing selection using disjoint selection techniques. *See also* disjoint selection.

extended selection list box

A list box that supports multiple selection, but is optimized for a selection of a single object or single range. *See also* extended selection and list box. *Compare* multiple selection list box.

File menu

A common drop-down menu that includes commands for file operations, such as Open, Save, and Print.

flat appearance

The visual display of a control when it is nested inside another control or scrollable region.

folder

A type of container for objects—typically files.

font

A set of attributes for text characters.

font size

The size of a font, typically represented in points.

font style

The stylistic attributes of a font, such as bold, italic, and underline.

gesture

A set of lines or strokes (inking) drawn on the screen that is recognized and interpreted as a command or character. *See also* recognition and ink.

glyph

A generic term used to refer to any graphic or pictorial image that can be used on a button or in a message box. *Compare* icon.

grayed

See unavailable.

group box

A standard Windows control that visually groups a set of controls.

handle

An interface element added to an object that facilitates moving, sizing, reshaping, or other functions pertaining to that object.

Help menu

A common drop-down menu that includes commands that provide access to Help information or other forms of user assistance. *See* also context-sensitive Help and task-oriented Help.

heterogeneous selection

A selection that includes objects with different properties or type. *Compare* homogeneous selection.

hierarchical menu

See cascading menu.

hold

To continue pressing a keyboard key, mouse button, or pen tip at the same location.

homogeneous selection

A selection that includes objects with the same properties or type. *Compare* heterogeneous selection.

hot spot

The specific portion of the pointer (or pointing device) that defines the exact location, or object, to which a user is pointing.

hot zone

The interaction area of a particular object or location with which a pointer or pointing device’s hot spot must come in contact.

icon

A pictorial representation of an object. *Compare* glyph.

implicit selection

A selection that is the result of inference or the context of some other operation. *See also* explicit selection.

inactive

The state of an object when it is not the focus of a user’s input.

inactive window

A window in which a user's input is not currently being directed. An inactive window is typically distinguished by the color of its title bar. *Compare* active window.

indeterminate

See mixed-value appearance.

ink

The unrecognized, freehand drawing of lines on the screen with a pen. *See also* gesture and ink edit.

ink edit

A standard Windows control for input and editing of “ink.” *See also* ink.

input focus

The location where the user is currently directing input.

input focus appearance

The visual display of a control or other object that indicates when it has the input focus.

insertion point

The location where text or graphics will be inserted (also referred to as the caret).

inside-out activation

A technique that allows a user to directly interact with the content of an OLE embedded object without executing an explicit activation command. *Compare* outside-in activation.

jump

A special form of a link that navigates to another location (also referred to as a hyperlink).

label

The text or graphics associated with a control (also referred to as a caption).

landscape

An orientation where the long dimension of a rectangular area (for example, screen or paper) is horizontal.

lasso-tap

A pen gesture that makes a region selection by drawing a circle around the object to be selected and tapping within that circle.

lens

See writing tool.

link

(v.) To form a connection between two objects. (n.) A reference to an object that is linked to another object. *See also* OLE linked object.

link path

The descriptive form of referring to the location of a link source (also referred to as a moniker).

list box

A standard Windows control that displays a list of choices. *See also* extended selection list box.

list view

A standard Windows list box control that displays a set of icons in different views (for example, a large icon, small icon, or list).

locale

A collection of language-related user preferences for formatting information such as time, currency, or dates.

localization

The process of adapting software for different countries, languages, or cultures.

marquee

See region selection.

maximize

To make a window its largest size. *See also* minimize.

MDI

See multiple document interface.

menu

A list of textual or graphical choices from which a user can choose. *See also* drop-down menu and pop-up menu.

menu bar

A horizontal bar at the top of a window, below the title bar, that contains menus. *See also* drop-down menu.

menu button

A command button that displays a menu.

menu item

A choice on a menu.

menu title

A textual or graphic label that designates a particular menu. For drop-down menus, the title is the entry in the menu bar; for cascading menus the menu title is the name of its parent menu item.

message box

A secondary window that is displayed to inform a user about a particular condition. *Compare* dialog box, palette window, and property sheet.

minimize

To minimize the size of a window; in some cases this means to hide the window. *See also* maximize.

mixed-value appearance

The visual display for a control when it reflects a mixed set of values.

mnemonic

See access key.

modal

A restrictive or limiting interaction because of operating in a mode. Modal often describes a secondary window that restricts a user's interaction with other windows. A secondary window can be modal with respect to its primary window or to the entire system. *Compare* modeless.

mode

A particular state of interaction, often exclusive in some way to other forms of interaction.

modeless

Not restrictive or limiting interaction. Modeless often describes a secondary window that does not restrict a user's interaction with other windows. *Compare* modal.

modifier key

A keyboard key that, when pressed, changes the actions of ordinary input.

mouse

A commonly used input device that has one or more buttons used to interact with a computer. It is also used as a generic term to include other pointing devices that operate similarly (for example, trackballs).

multiple document interface (MDI)

A technique for managing a set of windows whereby documents are opened into windows (sometimes called child windows) that are constrained to a single primary (parent) window. *See also* child window and parent window.

multiple selection list box

A list box that is optimized for making multiple, independent selections. *Compare* extended selection list box.

My Computer

A standard Windows icon that represents a user's private, usually local, storage.

Network Neighborhood

A standard Windows icon that represents access to objects stored on the network file system.

nondefault drag and drop

A drag (transfer) operation whose interpretation is determined by a user's choice of command. These commands are included in a pop-up menu displayed at the destination when the object is dropped. *Compare* drag and drop.

object

An entity or component identifiable by a user that can be distinguished by its properties, operations, and relationships.

object-action paradigm

The basic interaction model for the user interface in which the object to be acted upon is specified first, followed by the command to be executed.

OLE (Microsoft OLE)

The name that describes the technology and interface for implementing support for object interaction.

OLE embedded object

A data object that retains the original editing and operating functionality of the application that created it, while physically residing in another document.

OLE linked object

An object that represents or provides an access point to another object that resides at another location in the same container or a different, separate container. *See also* link.

OLE visual editing

The ability to edit an OLE embedded object in place, without opening it into its own window.

open appearance

The visual display of an object when the user opens the object into its own window.

operation

A generic term that refers to the actions that can be done to or with an object.

option button

A standard Windows control that allows a user to select from a fixed set of mutually exclusive choices (also referred to as a radio button). *Compare* check box.

option-set appearance

The visual display for a control when its value is set.

outside-in activation

A technique that requires a user to perform an explicit activation command to interact with the content of an OLE embedded object. *Compare* inside-out activation.

package

An OLE encapsulation of a file so that it can be embedded in an OLE container.

palette window

A secondary window that displays a toolbar or other choices, such as colors or patterns. *Compare* dialog box and message box. *See also* property sheet.

pane

One of the separate areas in a split window.

parent window

A primary window that provides window management for a set of child windows. *See also* child window and multiple document interface.

pen

An input device that consists of a pen-shaped stylus that a user employs to interact with a computer.

persistence

The principle that the state of an object is automatically preserved.

point

(v.) To position the pointer over a particular object and location. (n.) A unit of measurement for type (1 point equals approximately 1/72 inch).

pointer

A graphic image displayed on the screen that indicates the location of a pointing device (also referred to as a cursor).

pop-up menu

A menu that is displayed at the location of a selected object (also referred to as a context menu or shortcut menu). The menu contains commands that are contextually relevant to the selection.

pop-up window

A secondary window with no title bar that is displayed next to an object; it provides contextual information about that object.

portrait

An orientation where the long dimension of a rectangular area (for example, screen or paper) is vertical.

press

To press and release a keyboard key or to touch the tip of a pen to the screen. *See also* click.

pressed appearance

The visual display for an object, such as a control, when it is being pressed.

primary window

The window in which the main interaction takes place. *See also* secondary window and window.

progress indicator

Any form of feedback that provides the user with information about the state of a process.

progress indicator control

A standard Windows control that displays the percentage of completion of a particular process as a graphical bar.

project

A window or task management technique that consists of a container holding a set of objects, such that when the container is opened, the windows of the contained objects are restored to their former positions.

properties

Attributes or characteristics of an object that define its state, appearance, or value.

property inspector

A dynamic properties viewer that displays the properties of the current selection. *Compare* property sheet.

property page

A grouping of properties on a tabbed page of a property sheet. *See also* property sheet.

property sheet

A secondary window that displays the properties of an object when a user chooses its Properties command. *Compare* dialog box and property inspector. *See also* property page.

property sheet control

A standard Windows control used to create property sheet interfaces.

proximity

The ability of some pen devices to detect the presence of the pen without touching the pen to the screen.

push button

See command button.

radio button

See option button.

range selection

See contiguous selection.

recognition

The interpretation of strokes or gestures as characters or operations. *See also* gesture.

reference Help

A form of online Help information that can contain conceptual and explanatory information. *Compare* task-oriented Help and context-sensitive Help.

region selection

A selection technique that involves dragging out a bounding outline (also referred to as a marquee) to define the selected objects.

Recycle Bin

The standard Windows icon that represents the repository for deleted files.

relationships

The context or ways an object relates to its environment.

rich-text box

A standard Windows control that is similar to a standard text box, except that it also supports individual character and paragraph properties.

scope

The definition of the extent that a selection is logically independent from other selections. For example, selections made in separate windows are typically considered to be independent of each other.

scrap

An icon created when the user transfers a data selection from within a file to a shell container.

scroll

To move the view of an object or information to make a different portion visible.

scroll arrow button

A component of a scroll bar that allows the information to be scrolled by defined increments when the user clicks it. The direction of the arrow indicates the direction in which the information scrolls.

scroll bar

A standard Windows control that supports scrolling.

scroll box

A component of a scroll bar that indicates the relative position (and optionally the proportion) of the visible information relative to the entire amount of information. The user drags the scroll box to scroll the information. *See also* scroll bar shaft.

scroll bar shaft

The component of a scroll bar that provides the visual context for the scroll box. Clicking (or tapping) in the scroll bar shaft scrolls the information by a screenful. *See also* scroll box.

secondary window

A window that provides information or supplemental interaction related to objects in a primary window.

select

To identify one or more objects upon which an operation can be performed.

selection

An object or set of objects that have been selected.

selection appearance

The visual display of an object when it has been selected.

selection handle

A graphical control point of an object that provides direct manipulation support for operations of that object, such as moving, sizing, or scaling.

separator

An entry in a menu that groups menu items together.

shell

A generic term that refers to the interface that allows the user control over the system.

shortcut

A generic term that refers to an action or technique that invokes a particular command or performs an operation with less interaction than its usual method.

shortcut icon

A link presented as an icon that provides a user with access to another object.

shortcut key

A keyboard key or key combination that invokes a particular command (also referred to as an accelerator key).

shortcut menu

See pop-up menu.

single selection list box

A list box that only supports selection of a single item in the list.

size grip

A special control that appears at the junction of a horizontal and vertical scroll bar or the right end of a status bar and provides an area that a user can drag to size the lower-right corner of a window.

slider

A standard Windows control that displays and sets a value from a continuous range of possible values, such as brightness or volume.

spin box

A control that allows a user to adjust a value from a limited range of possible values.

split bar

A division between panes that appears where a window has been split; the split bar visually separates window panes.

split box

A special control added to a window, typically adjacent to the scroll bar, that allows a user to split a window or adjust a window split.

status bar

An area that allows the display of state information of the information being viewed in the window, typically placed at the bottom of a window.

status bar control

A standard Windows control that provides the functionality of a status bar.

stop

To halt a process or action, typically without restoring the state before the process began. *Compare* cancel.

submenu

See cascading menu.

System menu

See Control menu.

tab control

A standard Windows control that looks similar to a notebook or file divider and provides navigation between different pages or sections of information.

tap

To press and lift the pen tip from the screen, usually interpreted as a mouse click.

targeting

To determine where pen input is directed.

taskbar

The toolbar of the desktop. The taskbar includes the Start button, buttons for each open primary window, and a status area.

task-oriented Help

Information about the steps involved in carrying out a particular task. *Compare* context-sensitive Help and reference Help.

template

An object that automates the creation of new objects.

text box

A standard Windows control in which a user can enter and edit text (also referred to as the edit field).

thread

A process that is part of a larger process or program.

title bar

The horizontal area at the top of a window that identifies the window. The title bar also acts as a handle for dragging the window.

toggle key

A keyboard key that turns a particular operation, function, or mode on or off.

toolbar

A standard Windows control that provides a frame for containing a set of other controls.

toolbar button

A command button used in a toolbar (or status bar).

toolbar control

A standard Windows control designed with the same characteristics as the toolbar.

tooltip

A standard Windows control that provides a small pop-up window that provides descriptive text, such as a label, for a control.

transfer appearance

The visual feedback displayed during a transfer operation.

transaction

A unit of change to an object.

tree control

A standard Windows control that allows a set of hierarchically related objects to be displayed as an expandable outline.

type

(v.) To enter a character from the keyboard. (n.) A classification of an object based on its characteristics, behavior, and attributes.

unavailable

The state of a control whose normal functionality is not presently available to a user (also referred to as grayed, dimmed, and disabled).

unavailable appearance

The visual display for a control when it is unavailable.

undo

To reverse a transaction.

unfold button

A command button used to expand a secondary window to a larger size.

visual editing

See OLE visual editing.

well control

An inset field that is used to display color or pattern choices, typically used like an option button.

white space

The background area of a window.

window

A standard Windows object that displays information. A window is a separately controllable area of the screen that typically has a rectangular border. *See also* primary window and secondary window.

wizard

A form of user assistance that automates a task through a dialog with the user.

word wrap

The convention where, as a user enters text, existing text is automatically moved from the end of a line to the next line.

workbook

A window or task management technique that consists of a set of views that are organized like a tabbed notebook.

workspace

A window or task management technique that consists of a container holding a set of objects, where the windows of the contained objects are constrained to a parent window. Similar to the multiple document interface, except that the windows displayed within the parent window are of objects that are also contained in the workspace.

writing tool

A standard Windows pen interface control that supports text editing.

Z order

The layered relationship of a set of objects, such as windows, on the screen.

APPENDIX D

Level Extreme platform +
Subscription +
Corporate profile +
Products & Services +
Support +
Legal +
Français

Articles

Search: Search

How to change and how to reset mouse cursors (pointers)
Paul Vlad Tatavu, August 23, 1997
In Windows 95 and Windows NT you can use LoadCursor()/LoadCursorFromFile and SetSystemCursor() Win32 API functions to change or reset a system mouse cursor.

SUMMARY

In Windows 95 and Windows NT you can use LoadCursor()/LoadCursorFromFile and SetSystemCursor() Win32 API functions to change or reset a system mouse cursor.

DESCRIPTION

The following code shows how to do this correctly on Windows NT. It works also on Windows 95. This sample includes 3 functions: 1. CreategaSavedCursors() - It creates a global array (you can make it an array property to your application class if you prefer) where system cursors will be saved by the SetCursor() function. 2. SetCursor() - It saves the system cursor you want to change (if it was not saved before) and set the cursor to a new system cursor or cursor from file. 3. ResetCursor() - It resets a system cursor to the default/saved one. In order to simplify the code, I dropped any error checking and variable declaration. If you want to use this code, I strongly recommend you to include error handling code (you'll find some comments inside the code for basic error checking) and also, proper variable declaration.

```
function CreategaSavedCursors
*-- This function creates the gaSavedCursors global array.
*      1st column stores the standard cursors' ID's.
*      2nd column stores the saved cursor handle and
*          it's set by SetCursor function.
*-- WARNING: This function must be called only once and
*            before SetCursor or ResetCursor functions.

public gaSavedCursors[14,2]
gaSavedCursors[ 1,1] = 32650
gaSavedCursors[ 2,1] = 32512  && OCR_NORMAL
gaSavedCursors[ 3,1] = 32515  && OCR_CROSS
gaSavedCursors[ 4,1] = 32651
gaSavedCursors[ 5,1] = 32513  && OCR_IBEAM
gaSavedCursors[ 6,1] = 32648  && OCR_NO
*-- I couldn't find any documentation about this mouse pointer.
*   The value I use here is just a guess.
*   It works, but it may be wrong.
gaSavedCursors[ 7,1] = 32647  && ?
gaSavedCursors[ 8,1] = 32646  && OCR_SIZEALL
gaSavedCursors[ 9,1] = 32643  && OCR_SIZENESW
gaSavedCursors[10,1] = 32645  && OCR_SIZENS
gaSavedCursors[11,1] = 32642  && OCR_SIZENWSE
gaSavedCursors[12,1] = 32644  && OCR_SIZEWE
gaSavedCursors[13,1] = 32516  && OCR_UP
gaSavedCursors[14,1] = 32514  && OCR_WAIT
return


function SetCursor
parameter tnCursorID, txNewCursor
*-- tnCursorID is the cursor's ID you want to change
*   (The ID stored in the 1st column of the gaSavedCursors array)
*-- txNewCursor is the new cursor and it can be:
*       - a standard cursor ID (numeric)
*       - a cursor file name (character string)

*-- Declare Win32API functions to manage mouse cursors
declare integer LoadCursorFromFile in Win32API ;
                string
declare integer CopyIcon in Win32API ;
                integer
declare integer LoadCursor in Win32API ;
                integer, integer
declare integer SetSystemCursor in Win32API ;
                integer, integer

*-- Retrieve the cursor's index in gaSavedCursors array
for lnI = 1 to 14
   if gaSavedCursors[lnI,1] = tnCursorID
      exit
   endif
endfor

if lnI > 14
   *-- Incorrect tnCursorID
   return .f.
endif
```

```
if empty( gaSavedCursors[lnI,2])
   *-- The cursor was not saved before
   *   Save it!
   *-- LoadCursor returns >0 if successful
   lnCursorHandle = LoadCursor( 0, tnCursorID)
   *-- CopyIcon returns >0 if successful
   gaSavedCursors[lnI,2] = CopyIcon( lnCursorHandle)
endif

*-- Create the cursor handle
do case
   case type( "txNewCursor") = "C"
      *-- LoadCursorFromFile returns >0 if successful
      lnCursorHandle = LoadCursorFromFile( txNewCursor)
   case type( "txNewCursor") = "N"
      *-- SetSystemCursor will destroy the cursor that
      *   it takes as input parameter.
      *   So, use a copy of the needed cursor.
      lnCursorHandle = CopyIcon( LoadCursor( 0, txNewCursor))
   otherwise
      *-- Incorrect txNewCursor
      return .f.
endcase

*-- Change the cursor to the new one
*-- SetSystemCursor returns 0 if fails
lnError = SetSystemCursor( lnCursorHandle, tnCursorID)
return .t.


function ResetCursor
parameter tnCursorID
*-- tnCursorID is the cursor's ID you want to reset
*   (The ID stored in the 1st column of the gaSavedCursors array)

*-- Declare Win32API functions to manage mouse cursors
declare integer SetSystemCursor in Win32API ;
                integer, integer

*-- Retrieve the cursor's index in gaSavedCursors array
for lnI = 1 to 14
   if gaSavedCursors[lnI,1] = tnCursorID
      exit
   endif
endfor

if lnI > 14
   *-- Incorrect tnCursorID
   return .f.
endif

if empty( gaSavedCursors[lnI,2])
   *-- The cursor was not saved.
   *   Nothing to restore...
   return .f.
else
   *-- Restore the cursor
   lnError = SetSystemCursor( gaSavedCursors[lnI,2], tnCursorID)
   gaSavedCursors[lnI,2] = .f.
```

```
   return .t.
endif
```

Paul Vlad Tatavu

MORE ARTICLES FROM THIS AUTHOR

Finding the correct parameter for the CreateObject() function

Paul Vlad Tatavu, September 9, 1997

This is a fast solution when you're not sure about the parameter you should pass to CreateObject() for an ActiveX/OCX control that you know it is installed on your computer: 1. Open a form. 2. Click on the Ole Container Control button on the toolbar. 3. Draw a rectangle on the form for the new ...

Help Authoring Tools

Paul Vlad Tatavu, September 9, 1998

This is just a list of links to help authoring tools. The list is not complete and it is provided only as a startup help for those looking for such tools. I have not included technical characteristics/specifications because, as any other software, these tools are in a "permanent dynamical state" and...

How can I say if an object is a container?

Paul Vlad Tatavu, July 20, 1998

Determine if an object is a container.

How to be sure I am running only one instance of a VFP application

Paul Vlad Tatavu, February 8, 1998

Download the "Run only one instance of an application" file from the Files section, Classes-VCX category. The oneinst.zip file contains a VFP class that implements a mechanism do be sure the user is running only one instance of a VFP application.

How to change the full name for a user on Windows NT

Paul Vlad Tatavu, July 22, 1997

This code shows how to change the full name for a user on Windows NT using the Pointers Class.

How to get the flags for a user on Windows NT

Paul Vlad Tatavu, July 22, 1997

This code shows how to get the flags for a user on Windows NT using the Pointers Class.

How to get timezone information in Windows NT/95

Paul Vlad Tatavu, June 1, 1998

Please note that if the year (in the retrieved dates) is zero, than the dates are in day-in-month format. This format specifies a month (M), a day of the week (day_of_the_week) and a day (N) and it must be read as the M-th day_of_the_week in the M-th month. N can be 1 through 5 (5 means the last day...

How to register/unregister an OCX/ActiveX programatically

Paul Vlad Tatavu, January 17, 1998

There are two methods to do this: 1. Run the REGSVR32 program that is shipped with MS Windows. 2. Any OCX/ActiveX control, set of controls or OLE server is contained in a file that exposes 2 functions that can be used to register or to unregister the control(s) or the OLE server.

How to reset mouse cursors (pointers) on Windows 95

Paul Vlad Tatavu, August 14, 1997

This code shows how to reset the mouse cursors to the default cursor scheme set in Control Panel.

How to retrieve the local groups for a user on Windows NT

Paul Vlad Tatavu, October 28, 1997

This code shows how to use the Pointers Class to get the local group memberships for a user on Windows NT.

How to use CopyCursor() Win32 API function

Paul Vlad Tatavu, August 23, 1997

The CopyCursor() Win32 API function exists and it is documented in all Win32 API docs I could find. According all those docs, it can be used on Windows 95 as well as on Windows NT.

Some useful date functions

Paul Vlad Tatavu, July 12, 1997

Some useful date functions. The first day of the month specified by a date. The first day of the month specified by month and year. The last day of the month. The first day

of the week. The last day of the week. The number of months between two dates. The number of whole years between two dates. The...

The correct way of passing Unicode strings (or 2-byte character strings) to WIN/NET32 API functions

Paul Vlad Tatavu, June 14, 1997

In C/C++ language, ANSI strings (one-byte character strings) always end with chr(0) (null terminated strings). This is the string terminator and is the only indicator where a string ends. In FoxPro, chr(0) is not a string terminator.

Who opened what files on the network?

Paul Vlad Tatavu, December 7, 1998

The following program displays the open files, the users that opened these files and other related information. This code detects only the files opened using a net shared path. It does not return the files opened by a user on the local computer using a local path (i.e. the computer where the user is...

Copyright © 1993-2023 by Level Extreme Inc. All rights reserved · Website by levelextreme.com

APPENDIX E

Index

1. About this tutorial
2. Background
3. Introduction
4. How to start
5. Paint Counter: A windows program outline
6. Sort it out: A dialogue based Windows program
7. How do I do...

An Introduction to Windows programming

Extend your programming skills and get ready for up to date graphical user interface (GUI) programming

edited by:

Don D. Hobson

for Prairie View A&M University, February 1997

1. About this tutorial

This tutorial is designed to help those who have already some experience in PASCAL, C or C++ programming under DOS or UNIX to experience the fancy world of GUI programming. If you are to write programs not just for yourself but for other people you won't get around providing a nice and easy to use graphical user interface for your applications anyway, sooner or later. But even if you do not intend to write programs for a living this tutorial may give you a sound understanding of how modern operating systems and applications work.

Although many of the principles introduced here apply to all GUIs this tutorial will focus on Microsoft Windows since it is the most commonly used GUI. Windows also offers the most sophisticated development tools and there are plenty of books and other sources of information available. I will try not to use too many development terms here straight away but in order to understand what I am talking about you should have worked with Microsoft Windows and be familiar with common Windows terms like icons, scrollbars, dialogue boxes etc.

1.1 How difficult is it to write a Windows program?

Let's start with the good news: if you have already programmed in C or C++ before you don't need to learn a new programming language for programming Windows. But as you will see, programming Windows is somewhat different from what is called ANSI-C programming under DOS or UNIX. However different does not mean more difficult, although there are some hurdles to overcome at the beginning. But once you've understood the basics and written your first program many things get easier with Windows and you get opportunities you have never had before.

Many people think that its easier to do non-Windows programming first and do the "difficult" windows stuff later. This is probably one reason why windows programming is still paid too little attention in education where they "want to teach you the basics not the details". But programming Windows is more than just using a few more functions, its a different philosophy, its event driven rather than sequential and its not just coding but also designing and creating dialogue boxes and other program resources.

In the same way many also believe that you should understand C first before starting with object orientated programming in C++ say. I don't know where these myths come from and I can't even prove that they are just myths but what would you say if an alien asked you whether its easier to understand capitalism by learning about communism first. I would say no and I also think its pointless to start with "ordinary" programming when you really want to do Windows and its not necessary either to become a master in C first when you are aiming at C++ in the long run.

However the combination of Windows and C++ bears some difficulties for beginners. Using so called "class libraries" makes understanding of how Windows works more difficult as these class libraries hide a lot of functionality away from you. Defining your own classes is a bit tricky with Windows especially at the beginning due to a very lose link between Windows' objects and your own program objects.

This and the fact, that people who have experience in C cannot read C++ program whereas vice versa it is generally not a problem is why I will give you all the code samples in C. This tutorial won't make you a master Windows programmer anyway and you are welcome once you've

understood the philosophy of windows programming to go the much more promising way of C++ using either Microsoft's Foundation Classes (MFC) or Borland's Object Windows Library (OWL).

1.2 Software Development in the 90s

With the improvements in the way we use software, the way we develop it has also experienced considerable changes. Where a few years ago it was a challenge for a beginner to even type in a piece of source code from a book and compile it successfully we now get sophisticated so called Integrated Development Environments (IDE) that make things a lot easier for both the beginner and the expert.

Check you current way of programming. If you use an all purpose text editor to write your code and then call a compiler from a command line with something like ***comp -o myprog.c*** your not quite up to date. Even if you just want to taste what programming is like and especially if you are a beginner you might find features like source code highlighting, integrated source code debugging and full context sensitive help to name a few not just useful but crucial to get where you want to get. Writing a program is challenging enough and there's no need to fight your way though environment variables and makefiles.

In order to use this tutorial and to do windows programming all you need is a PC or at least access to a PC with Microsoft Windows and a Windows development package installed. The two most popular packages for developing Windows applications are Microsoft's Visual C++ and Borland's C++. Although both packages feature C++ compilers this does not mean you cannot write C programs with them. They both detect by default whether a source code file is a C or C++ file from the file extension which is .C for C and .CPP for C++ and compile them accordingly.

Which one of these packages you want to use is a matter of personal preference. I prefer Visual C++. With Borland you can also develop both 16-Bit Windows and 32-Bit Windows applications, whereas with Microsoft's Visual C++ you need Version 1.x (currently 1.52) to develop 16-Bit Apps and Version 4.x (currently 5.0) to create 32-Bit Apps. On the other hand Microsoft seems to have successfully establish its Foundation Classes (MFC) for C++ which make development especially of more advanced features of Windows such as OLE considerably easier.

Both packages include the following:

- Compiler and linker for both C and C++
- Integrated Development Environment IDE for project management, code writing and debugging
- A variety of other debugging and profiling tools
- Windows SDK including
- Library and header files for accessing the Windows API
- Online help files with the description of all API commands, messages, virtual key codes, etc.
- Resource compiler
- Tools to create Program resources such as Dialogue boxes, Menu's, Cursors, etc. (either Application Studio or Resource Workshop)
- Class libraries for easier C++ development (either MFC or OWL)
- Example code for C and C++ programs

1.3 How to use this tutorial

In this tutorial I will first give you some background knowledge about windows and then introduce you to some important terms and conventions. Then in section 4 we start with a first simple windows program which you can download to your machine. The files are compressed and put together into a single .ZIP file. To be able to use the files you have to extract them first. For every ZIP file you should create a new directory.

In order to load and compile the sample programs you need to create a project from the IDE. For that you will find a NEW command in the project menu in Microsoft Visual C++ 4.0. A project file keeps track of all the files belonging to a project as well as compiler and linker options necessary to build the executable and is basically a modern version of a makefile (don't worry if you do not know what that is). Microsoft's project files have the extension .mdp (for makefile) and they are still plain ASCII text files. However they should not be changed with a text editor (they've got a line at the beginning saying "DO NOT EDIT").

The tutorial is designed that you don't need any additional books however it is essential that as we go though the code you have the project loaded in your IDE and that you are able to access the online help of the Windows SDK. This is because it is neither possible nor sensible to explain all the functions we are using in detail with all their parameters and in the online help you will find all the information about a particular function or message you need.

Just in case you don't know: there are two ways accessing the online help from the IDE. First you will find a command called SEARCH in the help menu which takes you to the index page from where you can use the hyperlinks or the search command to find what you are looking for or just to get an overview of what is available. The other possibility is to position the cursor on a particular function name or data structure and press F1.

2. Background

This chapter discusses what makes graphical user interfaces so special and it gives you some background information about Windows and its current versions.

2.1 About being friendly to the user

When computers were invented they were used as number crunchers to solve mathematical or other scientific problems. The focus was clearly on the processing task but as computers evolved the input and output i.e. the communication interface between the computer and the machine got more and more important. In the first place this was necessary to process and visualise more complex data which could not be analysed in the form of lists of numbers. However the second major benefit was the easier way of using computers which allowed more and more people to use them.

User friendliness has since become one of the most important marketing issues and big software companies are investing millions of dollars in usability labs in order to improve the usability of their software products. Today, as algorithms and libraries for all sorts of technical problems are largely available, the design, coding and testing of the user interface is the most expensive and time consuming part in the development process of an civil end-user application. Unfortunately this aspect of computer science, which is in many ways more an art than a science in the classical way, is still very often paid too little attention in academia.

How can we measure user friendliness? Obviously since the actions undertaken by a user are unpredictable scientific methods are largely inappropriate. Looking at a single application there are three major key factors that determine user friendliness

1. **Intuitiveness**. This indicates how easy it is for the user to perform a desired task. Important factors are the grouping, ordering and naming of menu commands, the location, size and representation of buttons and the type of controls provided.

2. **Transparency**. This describes the type and accuracy of response given to the user before, while or after performing an action. Example: If an action undertaken by the user requires a complex computation which takes some time, the program can either not respond for the time the computation is in progress (bad), or display a "please wait" message (better) or show a progress bar indicating the percentage of computation finished (best). The latter is the most transparent to the user and thus most user friendly.Bubble help explaining the meaning of a button or status reports/ accurate error messages are other examples for good transparency.

3. **Restrictiveness**. The user friendliness also improves the less restricted the user is in his/ her actions. The user should e.g. be able to at any time cancel or undo operations and to customise the application's appearance and behaviour.

Taking the environment in which an application is operating into account user friendliness requires even more.

1. **Interoperability**. In a multiuser/ multitasking environment it is important that information can be easily exchanged between applications. The clipboard is a good example for such a feature, others are DDE and OLE.

2. **Standardisation**. This addresses a programs appearance and handling with respect to other applications. It requires that interfaces are the same or at least very similar for the same thing like e.g. dialogues for opening or saving a file or the configuration of the printer. The rule is: Once a user knows how to use one application he knows how to use them all.

2.2 Why Windows

The idea of a graphical user interface like windows is not new. In fact Microsoft started working on Windows as early as 1983 but the first two versions had little success to say the least. Apple was one of the pioneering companies that with the Macintosh introduced a Graphical User Interface Operating System for a mass market i.e. on an affordable personal computer system. And there are many similarities between the Mac-OS and Windows. On the less professional sector there were the Atari ST and the Commodore Amiga which first provided this type of advanced human computer interaction.

When Windows 3.0 came out in 1990 it was "as seven year overnight success" as Adrian King said in his book Inside Windows 95. Sitting on top of MS-DOS it was not the best technically possible solution for a GUI but it ran happily on ordinary PC hardware and the fact that you called it just like any other program let you the option whether to use it or not. And besides that there were in my opinion four other major reasons why Windows made it and is now dominating the computer world like no other GUI or operating system:

- Due to the 16 bit architecture Windows Programs and Windows itself were very moderate with memory requirements. Since four Megabyte of memory where more the exception than the rule at that time and it was thus also somewhat faster on slow machines than any 32bit Operating System would have been, stocking up hardware was not necessarily necessary. It also ran acceptably on Laptops and Notebooks which at that time were not monsters of performance and scaleability.

- There was also no urgent need for new software since old programs ran just as they did before without any loss of performance. This was crucial for the acceptance of both home users who in general do not have the money as well as for large businesses which could afford to update some PCs but were not willing to update a thousand. Not to mention the development costs for niche products.

- Microsoft did not just provide a GUI but also a state of the art Word Processor and Spreadsheet application. Its hard to tell whether Microsoft's Word and Excel were developed for Windows or Windows was developed for them because they reached the limit of

improvements they could do to their applications and wanted to get some competitive advantage over WordPerfect which was the best selling word processor at that time. Both Word and Excel were already available for earlier versions of Windows and Apple's Macintosh and they definitely set the requirements for Windows in a considerable way.

- Windows offered a consistent user interface. Consistent means that all programs appeared and behaved in the same way. If you have worked with Windows you probably know (but might have not consciously thought about it) that the first menu on the menu bar is always the File menu followed by Edit and View, no matter which program you are working with. And you automatically expect that the first menu item in the file menu is called New and under that there is the Open command. I could continue this list of features you just take for granted for quite a while but we will come across most of the later when we actually do some programming. This consistency is optional i.e. as an application programmer you don't have to stick to it, but if you want to write programs which will be used by other people you better do as Microsoft tells you. A consistent interface its one of the most important advantages Microsoft Windows compared to the old DOS Apps, and applies also to others Systems like the Mac-OS or NeXT-Step. This is also the reason why I personally truly dislike X-Windows the standard GUI for UNIX where nothing seems to be organised or standardised.

2.3 Windows != Windows

Since then Windows has developed and we now get three major versions of Windows: Windows 3.1 (including Windows for Workgroups 3.11), Windows 95 and Windows NT. Windows 3.1 is still fully depending on MS-DOS and thus based on a long outdated system architecture that Intel once designed for its 8086 processors. The programming interface for Windows 3.1 is called the Win16 API in opposition to the Win32 API offered by Windows 95 and Windows NT. Programming under Win16 is due different memory models and pointer constraints sometimes a bit tricky especially as programs get larger and more complex and many of the pitfalls can be avoided using the 32-bit programming model.

With Windows NT Microsoft got rid of all the compromises it had to make with MS-DOS and its 16 bit architecture and developed a fully independent operating system with ran exclusively in 32 bit mode. They also implemented a new file operating system called NTFS and network capabilities with the most sophisticated security features, a 3D graphics interface called OpenGL and even a new 16 bit character coding scheme (in opposition to the standard ASCI or ANSI character encoding schemes which feature only 8bit) called Unicode. Microsoft was aiming high to making Windows NT the state of the art operating system. Unfortunately this had its price: working with less than a 486-100 and 16MB of RAM was a pain and other system requirements were not moderate either.

Windows 95 is technically a compromise between the two, offering a 32 bit API but without all the advanced features of NT that I mentioned above. The major improvement of Windows 95 is the appearance and handling of the user interface which is somewhat different from the Windows 3.1 and Windows NT desktop. Coming versions of NT will also have this desktop interface.

Although there are considerable differences between all the three versions of Microsoft Windows the is good news for programmers: the source code is largely compatible. There are only a few API functions which underwent minor changes and the rest is just an extension of the API. Nevertheless you have still got to be a bit careful when converting a 16 bit program to 32 bit. One of the major changes is e.g. that the basic data type ***int*** which was 16bit wide in Win16 is now 32bit and there are no near pointers (16 bit pointers) any more.

Whereas Win16 programs can be used on all tree platforms, a Win32 program won't run directly under Windows 3.1. However there is a possibility to even do that with a operating system extension called Win32s. Provided that you do not use any API functions that are exclusively available under Windows NT or Windows 95 you can ship a single Win32 application for all three operating systems.

3. Introduction

This chapter introduces you to the basics of Windows programming and its event driven philosophy. It also explains why Windows programs look different from other C programs and introduces you to some major terms used in Windows programming.

3.1 Sequential programming vs. event-driven programming

It was only about a decade ago, when most computers had none or only very limited graphics capabilities. Interaction between the machine and the human was mainly done by typing in strange commands with a keyboard (and many self called UNIX Gurus still have not come over that yet) and receiving some sort of text message on the screen. Programs at that time had control over the interaction process as they gave the user a number of options what to do like pressing a key for example and they reacted by changing some of the say 2000 characters (25 lines by 80 columns) on the screen. Even if a program used a graphic mode this still applied, the only difference was, that instead of characters there were now a much larger number of pixels. So as a result there was a direct correlation of the way the computer saw the screen and the way the user saw it.

With GUI like Windows things have changed a bit. First you got this fancy device called mouse which enables you to go somewhere and click and even more important, you have now windows which you can drag around and arrange in any way you want. It would be a hard job for a program if it had to keep track of where it actually is and hence which of the thousands or millions of pixels it has to change to display a single character by also taking into account that some of the pixels may be hidden by another window. Someone's got to do the job though and this is where the GUI comes in. Now for a program that means that the correlation between the way the program sees the screen and the way the user sees it is lost. For the program there is only one window which is always visible and in which the coordinates of pixels never change.

This alone would not justify the need for a new programming philosophy because as you can see, many old non GUI programs still run happily in a "shell" window. However the reason why this works is that there is a windows program which suggests the old-style-application that the screen runs in text or full screen graphic mode and hence it does its output in the usual way. Since it would mess up the screen completely if the output would go directly to the screen the Windows program directs it to a buffer which it then displays on the screen. However this approach has the following disadvantages:

- Screen buffers require memory. For text-mode programs this is not a big issue since the amount of memory needed is only small but in graphic mode it can bring a system to its knees.
- The Windows program can't always know when a program changes the contents of the screen. So it has either to analyse the content of the buffer in order to detect changes or display the contents of the buffer very frequently. Both approaches are very time consuming and for graphical applications almost impossible to handle.

Another major problem is that although you might be able to change the size of such a window, it is not able to react to it i.e. if the program assumes a screen of 80x25 characters it won't give you more or less rows or columns just by changing the windows size.

All the old DOS or UNIX program did was processing every command sequentially. So apart from some programs which did their own interrupt handling (don't bother if you don't know what that is) it is always determinable with which command the program will succeed once the current command had been processed.

Under Windows things must obviously change a bit if we want to overcome the problems mentioned above. It can no longer be just us, telling the operating system or the GUI respectively what to do, here the OS must also be able to request services from us. Let me give you two examples of what that means:

- In a sequential C program you might use the **getc()** function to wait until the user presses a key on the keyboard. In Windows we will be told, whether we want it or not, that a key was pressed by receiving a message. We can then decide whether we want to process it or not, but we're not explicitly waiting for a key.
- If a Window or a portion of it becomes visible after being invisible (e.g. because another Windows covered it) we will be asked to redraw the window's contents or a portion of it. In a sequential program there is no need and thus no mechanism for such a thing as from the program's point of view all the information is always visible.

So as you can see the main difference between a DOS or UNIX program and a Windows program is that the latter cannot just sequentially process the code but must have a mechanism of reacting to events which may occur at any time. This type of programming is called event-driven programming. As a result it depends mainly on the user's actions which part of your program will get executed and the user is controlling the program rather than the program controlling the user. An event-driven program might e.g. have to redraw the window's contents five consecutive times without any changes because of some action undertaken by the user whereas in a sequential program there has never been a need to do that.

3.2 Some code for starters

If you have ever read a book about learning a programming language you have probably come across the fist program which printed "Hello World" on the screen. They always do that and so do most books about Windows programming although there is no such thing as a new programming language.

So in a conventional C program you have probably tried the following:

```
#include <stdio.h>

int main()

{

   printf("Hello World");

   return 1;

}
```

Now with Windows you expect things to be a bit more complicated and indeed in most books they give you about 50 lines of code or more to do that, although it can be as easy as

```
#include <windows.h>

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

{

   MessageBox(NULL,"Hello World", "My first program", MB_OK);

   return TRUE;
```

```
}
```

Although this is a fully working Windows program there is a good reason why you won't find it in the books: This is not what you really want. The ***MessageBox*** command displays only a simple message to the user and all the work for creating and displaying the window is done by Windows. The is no functionality you can add to a message box apart from the options provided in the arguments of the command and the window is owned by Windows rather than yourself. But what you really want is to have your own window which you can fill with more complex contents such as menu's, buttons, text and graphics and which the user can resize, minimise or maximise.

Before we do that however I want to analyse the above program to and make you familiar with some important conventions that are required for understanding Windows and our further exercises.

3.3 There is C and C

If you have done some conventional C or C++ programming and someone would ask you to name some of the commands in C you might say something like: **switch()**, **printf()**, **getc()**, **fopen()**, etc. Among those four command however there are two which cannot be used under Windows, one that you can but should not and only one that is a truly valid and acceptable command in a Windows C program.

In fact the latter three are commands which themselves call one or more functions in the OS and are thus specific to a particular platform. The implementation of these commands is provided in so called libraries which are linked together with your program, whereas the core C commands such as **switch()** can be understood directly by the compiler and thus do not require header or library files.

Under Windows you still use the same syntax, the same brackets and the same variable types as in a conventional C program and of course you need all the core C commands plus the so called preprocessor commands which begin with a hash (like e.g. #include or #define). Here is a list of the most common core C commands:

```
switch     case     break     if     else     while goto     return     sizeof     typedef     struct
```

However with other commands you have to be careful. Library commands which are all defined in header files like ***stdio.h***, ***stdlib.h***, ***io.h***, etc. cannot always be used under Windows. **Printf** (defined in stdio.h) is for example a command that you cannot use whereas **localtime** (defined in ***time.h***) you can. For some functions that can be used, you'll find a similar function in the Windows API as for example for **sprintf()**. **Sprintf** does the same as **printf** except writing the output to a string rather than the screen (or any other stream). **Sprintf** (defined in the ***string.h*** file) can still be used under Windows and you'll find it in many of even Microsoft's own sample code. However I recommend not to use it and use **wsprintf** which is the equivalent Windows function instead.

The trouble with library commands is:

- Library functions actually add code to your application and make it thus longer whereas Windows functions are just a function call.
- Windows does may things itself, like e.g. caching of file input and output. Windows API functions are optimised for that whereas Library functions like **fopen** (also defined in ***stdio.h***) sometimes do their own additional buffering and introduce a unnecessary additional level of complexity. Again memory is wasted for the buffer.

However there is one important difference between **sprintf** and **wsprintf**: **wsprintf** cannot deal with floating point numbers whereas **sprintf** can. So if you need to format floating point numbers you have to use **sprintf**.

If you got a bit confused now, here's a rule you can stick to: Before using a library function you should always check whether there is an equivalent or similar function in the Windows API and if so use it. Otherwise look up the description of the command in the manual or help file and check whether this command can be used under Windows.

3.4 The PASCAL calling convention

In every program things have got to start somewhere and in a conventional C program this is the function "main" which is executed by the OS when the program is called. Under Windows this function is called "***WinMain***" and it has got some additional parameters with it. If you look at the line of code with the implementation of the function ***WinMain*** you might realise a statement, that you have not seen before: PASCAL. You might have heard of the programming language PASCAL but since we're doing C does this make sense? It does and it actually tells the compiler that this function uses the PASCAL calling convention. Now what does that mean?

When a function is called by a program regardless whether it is an OS function or one defined in your own program, the parameters witch you specified in the function call have to be passed to the function. This is done by pushing these parameters on the stack and in C the order in which the parameters are pushed is from right to left (last parameter first) whereas in PASCAL it is vice versa. Not a big deal but as a result there are a couple of differences:

In C you can have functions with variable parameter lists as e.g. with **printf** where you can specify as many parameters as you wish (theoretically). With the PASCAL calling convention this is not possible but therefore functions using the PASCAL calling convention require 3 Bytes less of code when called (in 16-Bit programs). Now 3 Bytes may not seem very much, but by the time Microsoft designed Windows, Memory was a scarce resource and 3 Bytes saved for every of the thousands of function calls in Windows was a really big issue. 3 Bytes less mean also an improvement in speed since this also saves a few CPU cycles every time a function is called.

All of Window's API functions use the PASCAL calling convention except **wsprintf** because of the variable parameters required. And Windows expects, that all the functions it calls in your program also use the PASCAL calling convention. This is the reason why we have to specify the PASCAL keyword in the ***WinMain*** definition, otherwise the parameters would be messed up and you will most likely get a General Protection Fault (GPF) as soon a you use them. As we will see later there are other functions witch are called directly by windows such as window and callback procedures. These must in the same manner be declared with the PASCAL calling convention.

You might now ask yourself whether it is possible of using the PASCAL calling convention in general for all you functions and indeed most compilers offer an option to do that. However, I do not recommend to do so as you will get in trouble if you use other than the Windows API functions and which do not explicitly declare the calling convention in the associated header file. The compiler would then assume that these functions (which are provided in a LIB or OBJ file) also use the PASCAL calling convention and again you program would crash on execution. By default all C Compilers use the C calling convention an you should stick to that and only use PASCAL declaration where necessary.

3.5 The Hungarian notation

If you have a brief look at the above example again you might spot another difference between the two programs: the writing of the words. Where in the conventional C program you get only a rough idea if any at all what a command like e.g. **printf** does, the ***MessageBox*** command in the Windows program is much more understandable.

In the early days when C was invented programmers tended to be weird freaks, to lazy to write whole words and even too lazy to press the shift key. Programming commands were there sort of secret language and this might be one reason why many people have been afraid of programming.

In the Windows API all commands are concatenated full words in a natural language order and each of these words begins with a capital letter. To create a window e.g. the old C freaks would have probably introduced a command like **crwnd** or **wcreate** or **createwnd** whereas in Windows you can be sure to find a command called ***CreateWindow***. This helps you the programmer a lot to find a command that you have never used before. In many cases you just think what you would call the command you are looking for according to these rules and you will find such a command.

In Windows not just functions are named in a special way but also variables and structures. Every variable or function parameter starts with a lowercase type prefix abbreviation followed by the meaning in natural language with a capital letter. This way of naming variables is called the Hungarian notation in honour of an apparently legendary Microsoft programmer called Charles Simonyi (if you see him, give him my regards). The following table shows examples of how the Hungarian notation is used on variables:

Type Prefix	Data Type	Example	Description
b	BOOL	bEnabled	Boolean variable which can be TRUE (0) or FALSE (=0)
c	char	cKey	Single character
h	HANDLE	hWnd	Handle to a Windows Object
dw	DWORD	dwRop	Unsigned 32-Bit value
l	LONG	lParam	Singed 32-Bit value
w	WORD	wParam	unsigned 16-Bit value
u	UINT		unsigned integer value (can be 16 or 32 bit)
n	int	nHeight	singed integer value (can be 16 or 32 bit)
rc	RECT	rcWnd	Rectangle structure containing 4 coordinate values
sz	char[]	szFileName	Array of characters containing a String terminated by a character value of zero (zero terminated string)
lp	void far*	lpCallbackProc	Far pointer to something (e.g a hidden data structure) In 32-Bit Windows there are only far pointers so the type is (void *)
lpsz	LPSTR	lpszClassName	Far pointer to a zero terminated Sting
lprc	LPRECT	lprcWnd	Far Pointer to a Rectangle structure
lppt	LPPOINT	lpptPos	Far Pointer to a Point structure

When you define new data types or structure using the ***typedef*** statement all letters should be capital letters. This is e.g. how the types **BOOL** and **POINT** are defined in the ***windows.h*** header file:

```
typedef int BOOL;

typedef struct { int x,y } POINT;
```

Naming your variables and functions according to the Hungarian notation is not a must but I strongly recommend to adapt this style in your own programs. This will not only help others to understand your programs but also yourself it you look at them again after months or years. In fact I have not seen a single Windows program yet that does not use this notation.

3.6 Handling Windows objects

Now we've got all this fancy windows, icons, fonts etc. stuff but how do we handle it? Well, with handles of course. In the real world e.g. most glass windows have a handle and you use it to open and close it. In a windows program it basically the same thing. When you create the window you get a handle which you can then use to show, hide, move, size or destroy the window. And thanks to the Hungarian naming convention the functions are called respectively ***ShowWindow***, ***HideWindow***, ***MoveWindow*** (used for both moving and sizing) and ***DestroyWindow***. All these functions require the handle of the window you want to perform the action on as the first parameter.

However window handles are just one particular type of handles and there are a lot more types in the Windows API. You get handles for all types of windows objects such as icons, bitmaps, dialogues, menus, fonts, color palettes and so on.

Now what exactly is an handle? Again if you have done some conventional programming you might have already come across one type of handles: file handles. A file handle is a positive integer value returned by either *open* or *create* and it identifies a file. Every time you read of write to/ from the file you need to specify the file handle and when you finished you *close* the file and the file handle becomes invalid.

In MS-DOS and UNIX the value of a file handles is an index number to a table owned by the OS where information about the file is stored. Since you cannot access the table the value of the file handle is more or less meaningless to you and it can only be used to call other file functions. Under Windows handles are usually pointers to a data structure describing the associated object and in the same manner it is meaningless to the application. However since the value of a handle is unique for a particular object type (i.e. there won't be two different objects of the same type with identical handles) you can compare two handles and if they have the same value they refer to the same object.

A common mistake in windows programming and the cause for many general protection faults (GPFs) is to pass an invalid handle to a function. A handle is only valid as long as an object exists. A window handle e.g. becomes invalid either if you explicitly call ***DestroyWindow*** in your program or if the user closes the window. If a handle becomes invalid you should set its value to NULL to be able to detect whether is valid or invalid later on.

3.7 Resources

In the real word nothing is unlimited and this also true for the computer world. For all those who own a car, parking space is (apart form money) probably one the things that is most limited and hard to find these days. In general we call things like that Resources.

What exactly is a resource? Well first it is something that is given by nature (or something/ someone else that beyond our control) i.e. it can't easily be produced, second it is limited and third we can decide when and how much we want to use of it. Real word resources are as we all know oil and coal for example.

When we look at the computer world **hardware resources** are the most obvious. There is a given amount of physical memory, the CPU operates at a given speed and there is a certain amount of hard disk space available. Obvious isn't it, but have you ever looked at the screen a resource? Well its properties are given by the manufacturer of your monitor and you can't easily change them, it is limited, as there is a maximum number of horizontal and vertical pixels and it is used up by all these programs that you simultaneously run in your GUI environment.

On the software bit there are also resources but they are not quite as easily to spot. First there are the so called **System Resources**. These are normally linked to hardware resources in some way like for example the available memory (but here its physical + virtual memory). Under Windows there are some other limits which are especially a problem under Windows 3.x and were divided into KERNEL, GDI and USER Resources, which all refer to subsystems of Windows. There is e.g. a limit on the number of file handles that can be obtained and that means that only a certain number of files can be opened simultaneously. Since IO operations are handled by the Kernel subsystem, this is a Kernel resource. GDI resources are limits set by the graphical subsystem of Windows and it actually means a limit of the memory available to store information about graphical objects as pens, brushes, bitmaps etc. Note that the limit only refers to the information stored about an object not the object itself i.e. for bitmaps that is amongst other things the size in pixels and a pointer to the bitmap data but not the bitmap data itself. The limit is also not set for each object type individually but for the sum of all GDI objects. So if the limit was 10 objects there could be either combination of the number of pens, brushes and bitmaps up to the total limit of 10. GDI resources were normally the most likely to run out of under Windows 3.1 and once you ran out, you were lucky if you were able to save your data and reboot windows. Finally USER resources are the number of Windows classes and individual windows that can be defined and opened throughout the system. As windows come and go there is very rarely a deficiency so this is generally not a problem to worry about.

As I mentioned before this is a particular problem of Windows 3.x. Under Windows NT there is no preallocated memory for resources and thus the only limit it the total memory available which will lead to other problems first once you are running out. Windows 95 is again somewhere in between. Technically its still the old inflexible system of Windows 3.x, but they have split resource types up and use more resource heaps, which basically means that it takes longer before a shortage occurs.

Since now resources have been a pain so let's now get on to a more friendly type of resources, the so called **Program resources**. Program resources are any kind of no-code information you create yourself in order to add it to your program. Examples are tables of any kind, bitmap

images, icons, cursors, dialogue boxes etc. We'll go into more detail later on, when we create program resources.

From your point of view these might not look like resources but from the programs point of view they are since they are

- given (by you, the programmer)
- limited i.e. there is only the ones you have previously defined and they cannot be modified or extended at runtime (although they are not used up)
- the program can use them when and as much of them as necessary.

In conventional DOS and UNIX program resources are normally stored in external files which you have to take care of yourself. In Windows program resources are linked together with your EXE file and there a plenty of functions to access and use these resources.

3.8 Compiling the lot

Obviously the most important thing you need to compile a Windows program is at least one C or CPP (C++) file containing your code. In addition every "real" Windows program needs another file type: a resource file with the file extension .RC. This is where you put all your program resources in such as e.g. menus and dialogue boxes and the first thing that should go in there is a program icon. Without an icon, Windows will use a default icon which gives little information about your application when installed in the Program Manager or the Windows 95 Desktop.

Most Windows projects contain another file, the definition file with the extension **.DEF**. This is required for 16 Bit Windows applications and optional for Win32. It contains general information about the program and sets some initial runtime parameters. **DEF** files are normally created at the beginning of the project and are hardly changed later on. If you set up a new project just copy and existing **.DEF** file and modify it according to your requirements.

Now what happens with all these files when you compile them? The following graphic shows what happens with your source files when compiling:

In the first stage all your code files are compiles into OBJ files. Then the resource compiler translates your RC file into a RES file. This is the binary representation of your resource description and contains also files included in the RC file such as .ICO and .BMP files.

In stage 2 all object files are linked to an EXE file. This is a temporary executable file which cannot be executed. In the next stage the resource compiler links the RES file with the temporary EXE file and marks it as a Windows program.

4. How to start

After all this theory you might feel like doing something practical now or at least to have a look at some real code. So how do we start with a windows program? Actually there are two different approaches.

- The first one is the "ordinary" way of doing it. This is the way you will find in most books about Windows programming and it requires almost exclusively coding (C or C++). This approach features registering a window class and creating a window derived from that class. Here you will learn a lot about message queues and client areas. The problem with this approach is, that it takes you a bit longer to get somewhere and if you are not that keen on coding you might have a look at the other option.
- The second approach is a more interactive way of designing an application and coding is here only the second thing in line. This approach is especially suitable for simple programs but there are no limits really. If you were to create something like the calculator you get with windows say, your definitely better of with this approach to start with. This approach is based on a dialogue box which you design with a special interactive tool. Afterwards you just add some code for each of the control elements you used in your dialogue.

In fact almost every windows program features at least one dialogue box and so you need the knowledge of approach two anyway. And vice versa if you want to extend your dialogues with say you own custom controls you need the knowledge of approach one. In fact a complex windows program is always a mixture of both techniques. However I reckon that even many people with Windows programming experience are not aware of the fact, that almost every application can be completely based on a dialogue box, simply because it is not in the books.

So how far do we get with say 200 lines of code?

200 lines of code is not much and you'll be surprised how far you'll get. The next two chapters give you an outline for a windows program based on each of the two approaches. You don't have to understand both approaches straight away. See which one you prefer and try to modify and extend it a little bit. You can always come back to the other approach later. Finally in chapter seven I will then give you some ideas on how to build on the knowledge you have acquired by giving you a free choice on where to go next.

- Chapter 5: Paint Counter: A windows program outline Based on approach 1, this example shows you create a simple windows program that counts paint messages of a window.

- Chapter 6: Sort it out: A dialogue based windows program Based on approach 2, this sample application allows to add names to a list and sort it.
- Chapter 7: How do I do... The "how do I do" page gives you hints and clues on how you can you extend your programs and where to get further information from.

5. Paint Counter: A windows program outline

In this chapter we analyse the code required for a simple windows program based on a custom window class. The purpose of the program is to show you how message are distributed and graphical output is performed. The only thing the program actually does is to count the number of paint messages the window is receiving in order to demonstrate which events cause a repaint message.

5.1 The files

To compile and modify the program you need the following files:

Example1.C - the main code file

Example.RC - the main resource file

Example1.h - Header file

`Example1.ICO` - Icon file containing program icon in binary form

`Ex1W16.EXE` - 16-bit executable file for Windows 3.x

`Ex1W32.EXE` - 32-bit executable file for Windows 95/ Windows NT

Example1.zip Click here to download the archive to your computer.

5.2 The procedure WinMain

Let's have a look at the program entry point, the procedure ***WinMain***. This is where it all starts and what we get is four parameters.

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpCmdLine, int nCmdShow)
```

The first parameter **hInst** is a handle for the current instance of our program. Now what do we mean by instance? As you know, you can start most windows programs multiple times and you will get another window every time. Take Notepad e.g. and start it 10 times say. You've now got 10 instances of Notepad and each of them are independent from each other. Internally Windows loads the program's code only once in order to save memory space, but it gives every instance its own data area and stack and you do not have to bother about whether there is one or many instances of your program.

This handle is very important as we will need it later on as a parameter for various functions and so we store it in a global variable.

```
hInstance=hInst;
```

Many people would tell you its bad to use global variables at all and generally I agree. Global variables cause a number of problems and make programs less understandable and maintainable. In this case however its different. Basically this is because it is not really a variable but more of a constant, since it will not change at any time. Unfortunately C does not offer a feature to set the value of a variable only once but since we set it right at the beginning and we know it wont change all we have to make sure is that we do not change it by mistake.

The second parameter of ***WinMain*** is **hPrevInst**. Under Win16 this is a handle to a previous instance, if any, otherwise it is NULL. Basically this handle indicates in which memory segment the data of the previous instance is located. Under Win32 things have changed a bit. First there is no memory segmentation any more and second instances are more protected against each other which makes this parameter obsolete and it will thus always be **NULL**. The only function that makes use of this handle under Win16 is ***GetInstanceData***, but since there is no such handle in Win32 it has been deleted and so its better to forget about it straight way.

However in Win16 this handle has another function - the detection of the first instance of a program which is the case when ***hPrevInst*** is **NULL**. This is important because only the first instance must register user defined window classes that the program wants to use (we will come to that in a minute). Again in Win32 a program does not know whether it is the first instance and thus it must always register the window classes required.

We use the following code to detect whether we are the first instance and if so we call our own function to register the window classes. Since in Win32 ***hPrevInst*** is always **NULL** this will also work fine for 32-bit programs. A description of what ***RegisterWindowClasses*** does can be found in the following paragraph.

```
if (hPrevInst == NULL)

      if (!RegisterWindowClasses(hInstance))

           return FALSE;  // registration failed!
```

After the window class has been registered it is now time to create our application window. For that we use the function ***CreateWindow*** and if the window can be created as specified we will receive a window handle in return. The ***CreateWindow*** function requires a number of parameter which I will not explain in detail since a precise description is provided in the online help. The window will have the title "My first Windows Program" and its size will be 400 by 300 pixels. With the **CW_USEDEFAULT** parameters we leave it up to Windows where on the screen the window will be shown later on.

```
// Create our Main Application Window

hAppWnd=CreateWindow(szWndClassName,"My First Windows Program",

                     WS_OVERLAPPEDWINDOW|WS_HSCROLL|WS_VSCROLL,

                     CW_USEDEFAULT,CW_USEDEFAULT,400,300,

                     NULL,NULL,hInstance,NULL

                    );
```

As it is always good practise to check the return value of API functions we now check whether the window handle is valid. If it is not i.e. if the window handle is **NULL** we terminate the program by returning FALSE.

```
if (!hAppWnd) return FALSE; // Create Window failed
```

The window is now created but it is not visible on the screen yet (Note: if you want a window to be visible immediately after creation add the window style **WS_VISIBLE**). To display the window we call ***ShowWindow*** which requires the window handle and a flag indicating how the window should be shown. The latter can be for example **SW_SHOWNORMAL** if the window should be shown in its given size, **SW_SHOWMINIMIZED** or **SW_SHOWMAXIMIZED** if it should be initially an icon or full screen or any other of the SW_ constants defined in ***windows.h*** (see function ***ShowWindow*** in the SDK help file).

For our main window however we should not set this ourselves but use the parameter ***nCmdShow*** which is the last parameter of the ***WinMain*** procedure. Usually this will be set to **SW_SHOWNORMAL** but the user can specify otherwise in the program manager. Afterwards we force the window to paint its contents by calling ***UpdateWindow***.

```
ShowWindow(hAppWnd,nCmdShow);
UpdateWindow(hAppWnd);
```

Now that the window is created we come to the most important part of the ***WinMain*** procedure: the message loop. This is a simple while loop that will poll message form the message queue and dispatch them to the associated windows for further processing.

```
while (GetMessage(&msg, NULL, 0, 0))
  {
     TranslateMessage(&msg); /* translates virtual key codes    */
     DispatchMessage(&msg);  /* dispatches message to window    */
  }
```

There is another parameter in the ***WinMain*** procedure that I have not talked about yet. ***lpCmdLine*** is a pointer to a zero terminated string with the command line parameters specified in the program manger of Win 3.x or the shortcut properties of Win 95 respectively. Unlike in DOS or UNIX programs the parameters are not split up automatically i.e. if you call your program e.g. with "**C:\TEST\MYPROG.EXE -f hello.tst -n**" ***lpCmdLine*** will point to a string containing "**-r hello.tst -n**".

Although command line options are pretty unimportant in a GUI environment anyway there is one thing the ***lpCmdLine*** should always be used for: if your program deals with files you should always check whether ***lpCmdLine*** contains a file name and if so open it. This is important since many users start programs by double clicking on a document file in the file manager or the explorer. Provided that the file extension of the document file is assigned to your application, these programs will then call your program and specify the documents file name in the ***lpCmdLine*** Parameter.

5.3 Classifing a Window

In the description of the ***WinMain*** procedure above I have missed to explain what ***RegisterWindowClasses*** actually does. This function is defined next in the C file as follows:

```
static BOOL RegisterWindowClasses(HINSTANCE hFirstInstance)
```

This function, which is called from the ***WinMain*** for the first instance in a Win16 program and for every instance in a Win32 program, requires a handle to the instance and returns a boolean parameter indicating whether the function was successful (return value is **TRUE**) or not (return value is **FALSE**). I also added the keyword static which makes the function invisible i.e. inaccessible outside this module. It is good practise to declare every function that is not called from outside as static as you can find out more easily where this function is called from and you can use the function name again in other modules.

Lets just discuss what a window class is. A window class defines general properties for all windows that are derived from this class i.e. once a class has been defined you can create any number of windows based on this description and they will all have certain things in common like the background color, the icon that is shown when the window is minimised or most important the Windows procedure which handles all events concerning the window.

In order to register a window class we need to fill a structure of type **WNDCLASS** which is defined in ***windows.h***. For that we define a local variable of this type and set the name of the new window class.

```
{ WNDCLASS wc;

  wc.lpszClassName = szWndClassName;  // Name of the Window Class
```

The name we set here for our class can be virtually any name that takes our fancy except the name of predefined window classes. Predefined classes are BUTTON, STATIC, LISTBOX, COMBOBOX, SCROLLBAR and EDIT which are the names of all standard controls provided by Windows. The class you register will be available for your application only and be invisible to others; hence you also do not need to worry whether the name you are giving your class has already been taken by another application.

The name is not given here directly but it is hidden in the constant string array ***szWndClassName*** which I have defined at the beginning of the file as:

```
static const char szWndClassName[]={ "MYWNDCLASS" };
```

Hence the name of the class we are registering is **MYWNDCLASS**. We need this name also for the function ***CreateWindow*** which is the reason why I have put it in a variable. Again as we only need the string in this module it is saver to define it as static which makes it invisible to other source code modules.

Next we set the instance handle and some class style flags. A concise description of possible class style flags can be found in the online help. The style I used here forces the window to redraw completely every time the size of the window changes. You might want to try out what effect it has if you do not specify these flags. For that set ***wc.style*** to zero and recompile the program.

```
  wc.hInstance = hFirstInstance;         // Handle of program instance
  wc.style     = CS_HREDRAW|CS_VREDRAW;  // Combination of Class Styles
```

Now we need to specify the address of a function handling all the events concerning the window. This function is called the windows procedure and you'll find the body for the function in the following paragraph. Since all windows derived from this class share the same window procedure we have a problem if more than one window per program instance is derived from this class. How can we give each window its own private data? The answer lies in the ***cbWndExtra*** parameter which gives the number of bytes allocated for each window to store user defined data. In C++ this is e.g. used to store a pointer to a C++ object. To set and retrieve the data Windows offers the functions ***SetWindowWord*** and ***SetWindowLong*** or ***GetWindowWord*** and ***GetWindowLong*** respectively.

```
  wc.lpfnWndProc   = AppWndProc; // Address of Window Procedure
  wc.cbClsExtra    = 0;   // Extra Bytes allocated for this Class
  wc.cbWndExtra    = 0;   // Extra Bytes allocated for each Window
```

The rest of the structure members defines the basic appearance of the windows created from this class. First we load the Icon that is displayed when the window is minimised from the resource file. ***LoadIcon*** does the job by returning a handle to the icon. To identify the resource we use the macro **MAKEINTRESOURCE** which is also defined in **windows.h** and which takes a numeric value uniquely identifying a resource. Its up to us to assign numbers or names to resources so I have defined **ICON_APPWND** in the ***example1.h*** header file as:

```
#define ICON_APPWND 100
```

If you want to have your own cursor when the user is in the window you can use ***LoadCursor*** accordingly. In this case we use one of Windows' standard cursors which is why we set the instance handle to **NULL**. Finally we set the background colour to white and the menu name to **NULL**.

```
  wc.hIcon         = LoadIcon(hFirstInstance, MAKEINTRESOURCE(ICON_APPWND));
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName  = NULL;
```

Now that all members of the structure describing the window class have been set we register the window by passing a pointer to the structure on to windows.

```
   if (!RegisterClass(&wc))  /* Register the class */
       return FALSE;         /* RegisterClass failed */
```

When your program gets more complex you might need to register other windows classes.

5.4 The window procedure

Where all the code we have looked at so far is executed in the first few milliseconds after program execution we come now to the heart of your program, the windows procedure. As I mentioned before this is where Windows sends all events concerning the window to and where we can decide how to react to them. This is also the place where you will extend your program in order to add more functionality.

Let's have a look at the general structure of a window procedure first:

```
LRESULT FAR PASCAL WndProc(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
  switch(msg)
   {
     case WM_...
          /* process a message */
          break;
     default:
          /* let Windows handle the message */
          return DefWindowProc(hWnd,msg,wParam,lParam);
   }
  return 0L;
}
```

As the window procedure is called by Windows it must always be defined as PASCAL i.e. it must use the Pascal calling convention. The keyword FAR is important for Win16 only but its no mistake to specify it for Win32 either. The window procedure receives four parameters from Windows and returns a 32 bit integer value.

The first parameter is the handle of the window which is exactly the one we received from the ***CreateWindow*** procedure in the ***WinMain***. The second parameter is a message number indicating the type of event that has occurred. The value and meaning of the last two parameters are depending on the message.

All window messages are defined in ***windows.h*** and they all begin with the prefix **WM_**. There are hundreds of messages and you will never need to process all of them in a window procedure. Here are some of the most often used which you can look up in the help file.

```
WM_CREATE            WM_PAINT             WM_DESTROY           WM_SIZE
WM_LBUTTONDOWN       WM_ACTIVATE          WM_COMMAND
```

All messages that you do not process must be passed on to the default message handler of windows. To do that the function ***DefWindowProc*** is called in the default branch of the switch statement.

Now let's look how the Window procedure for our sample application has been implemented:

```
LRESULT FAR PASCAL AppWndProc(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
  static int iPaintCount;   // Count the number of Paint Messages received
  static HBRUSH hFillBrush; // handle to a brush which we use for filling the ellipse
```

First we define a variable which we use later on to count the number of **WM_PAINT** messages that have been received. This variable must be defined as static in order to preserve its value when we return control to Windows. Otherwise its value would be some random number every time Windows calls the window procedure. And we need another static variable to store the handle of the brush we want to fill the ellipse with.

Now we filter the messages and the first message we're going to process is **WM_CREATE**:

```
   switch(msg)
    {
      case WM_CREATE:
           iPaintCount=0;
           hFillBrush=CreateSolidBrush(RGB(0,255,255));   // Create a cyan brush
           break;
```

This message is received only once and that is when the window is created. We receive this message from Windows as a result of the call to ***CreateWindow*** we made in the procedure ***WinMain***. So at this point, the program will still be in the ***WinMain*** waiting for ***CreateWindow*** to return, which will only be the case after we finished processing our **WM_CREATE** message. This message is now used for initialising variables and data structures and allocating all kinds of resources required by the window. In our case we set the counter variable to zero and create a brush that we will use for filling the ellipse. From ***CreateSolidBrush*** we receive a handle to a brush of the desired color which is specified in terms of its red, green and blue content using the ***RGB*** macro. This handle is a resource allocated by windows on behalf of your application and must be freed later on using ***DeleteObject***.

The next message in line is the **WM_COMMAND** message. It deals with menu commands only. By examining the ***wParam*** parameter we can determine which menu command the user selected. All menu commands are associated with a number which is defined in the ***example1.h*** file; all beginning with **IDM_** standing for IDentifier Menu.

```
case WM_COMMAND:
     switch(wParam)
      {
        case IDM_ABOUT:
             DialogBox(hInstance,MAKEINTRESOURCE(DLG_ABOUT),hWnd,AboutDlgProc);
             break;
        case IDM_QUIT:
             DestroyWindow(hWnd);
             break;
      }
     break;
```

There are only two commands in the menu the *About* and the *Quit* command. If the user decides to quit, we simply destroy the window using ***DestroyWindow***. The **WM_DESTROY** message will then do everything else that is necessary. If the user selected the about command the ***DialogBox*** command is used to display a windows defined in the resource file. This is described in more detail in the Dialogue Example.

In order to force a repaint of the window we allow the user to press the left mouse button. This is again easy to detect as we will be sent the following message:

```
case WM_LBUTTONDOWN:
     // Invalidate the window's client area
     InvalidateRect(hWnd,NULL,FALSE);
     break;
```

There are other messages similar to that indicating other mouse events such as **WM_LBUTTONUP**, **WM_MOUSEMOVE** etc.

What we do here is not repainting the window directly but rather invalidating the window's client area. The client area is the area of the window inside its borders i.e. all the space that is at your disposal and that you are responsible for drawing. ***InvalidateRect*** takes three parameters: the handle of the windows which client area you want to invalidate, the part of the client area you want to invalidate, and a flag indicating whether the invalidated parts should be erased first, or whether you want to paint over the existing contents. The part you want to invalidate is specified by a pointer to a rectangle containing the boundaries but is here (and in most cases this is absolutely sufficient) set to NULL which means that the entire client area is invalidated.

Now how does that help us? ***InvalidateRect*** will add a rectangle to the window's update region and post a **WM_PAINT** message to the window. If many events happen simultaneously each requesting to repaint the window, then if you'd do it directly you might end up, drawing the same thing several times. With this mechanism all requests are collected and processed later. Additionally this also takes into account, which part of the window is visible in case of overlapping windows. Window's will automatically minimise the area that must be repainted.

Now there has got to be some code to actually draw the contents of the window and the time and place for that is the **WM_PAINT** message. This message is received first when the window becomes visible and then every time that Windows wants us to repaint the entire window or a portion of it, which can virtually be any time. Since the whole purpose of the program is to give you an idea when and how often this message is received we increase the value of our counter every time and display its value. Now how do we do that?

```
case WM_PAINT:
     { PAINTSTRUCT ps;
       HDC hdc;
       // Increase the Paint Message Counter
       iPaintCount++;
       // Get the handle to the Windows's Display Context
       hdc=BeginPaint(hWnd,&ps);
        // Now Paint the Windows's contents
        PaintAppWindow(hWnd,hdc,iPaintCount,hFillBrush);
       // Call EndPaint to release the DC and validate the client area
       EndPaint(hWnd,&ps);
     }
     break;
```

First we allocate a structure of type **PAINTSTRUCT** which we let Windows fill with values by calling ***BeginPaint***. The meaning of the structure members is of minor importance at the moment and what we're really after is the handle to the display context (this is what **HDC** stands for). This

handle is given to us in return to the call to ***BeginPaint*** and it is actually also contained in the **PAINTSTRUCT** structure. Fair enough, you might say, but what the heck is a display context?

In brief a display context is something that tells Windows where any output the window wishes to make has to go on the screen and where the borders of the area are. In fact you won't be able to draw or write anything in your window without having a valid handle to such a display context. All output is managed by the Graphic Device Interface (GDI) subsystem of Windows and I will tell you more about display contexts and the GDI in the next section.

So now that we've got the handle to display context we can actually draw our message which we do using the function ***PaintAppWindow*** which I will explain in the next paragraph. After we have done that, we must call ***EndPaint*** which tells Windows that we have finished painting and that the contents of the window are now in a valid state. End Paint will also free the display context i.e. the handle becomes invalid and we can't use it any more.

Processing the **WM_PAINT** message and calling ***BeginPaint*** and ***EndPaint*** is the single most important thing a window procedure has to provide. However messing around with it can cause a lot of trouble so I better warn you of some of the most dangerous pitfalls a beginner can step into:

- ***BeginPaint*** and ***EndPaint*** must only be called in response to a **WM_PAINT** message. If you wish to do painting in response to other messages you have to use ***GetDC*** and ***ReleaseDC*** to obtain and free the handle to the display context.
- Do never even think about keeping the handle to the display context by not calling ***EndPaint***. There is a limit of five display contexts that are available at a time and not calling ***EndPaint*** before returning from the **WM_PAINT** will almost inevitably lead to a mess on the screen and the need to reboot Windows.
- Do not call API functions which may result in receiving other messages or the direct repainting of other windows between ***BeginPaint*** and ***EndPaint***. You are on the save side if you only use Windows functions which require a handle to a display context such as ***DrawText*** or ***LineTo*** (all of these are GDI functions). Functions to avoid are those which require a window handle such as ***UpdateWindow*** or ***MoveWindow*** (which are functions of the USER subsystem). If you want to call one of these functions in response to a **WM_PAINT** message you can always do it before ***BeginPaint*** or after ***EndPaint***.

The last message we have to process for our example application is the **WM_DESTROY** message and you probably guess that this is sent when the window is destroyed.

```
      case WM_DESTROY:
           DeleteObject(hFillBrush); // Delete the fill brush (never forget!)
           PostQuitMessage(0);       // Tell Windows we want to terminate
           break;
```

First we destroy the brush we allocated in the **WM_CREATE** message. If you have more resources allocated in the **WM_CREATE** message such as memory or file handles, this is the place and time to free them.

Now Windows does not know, that the destruction of this window means the end of our program. So what we have to do here in order to tell Windows that we want to terminate our application is to call ***PostQuitMessage***. Window will then put a **WM_QUIT** message in our message queue and that will give our call to ***GetMessage*** in the procedure ***WinMain*** a return value of zero. If you look at the message loop we have defined in the ***WinMain*** again you will see that a return value of zero terminates the loop and the procedure ***WinMain*** returns. Now none of our code is executed any more and Window will remove our application from memory. Failing to call ***PostQuitMessage*** would keep our application waiting for a message but since there is no window any more there won't be any more messages. Unfortunately the only way to get rid of such an application (if you do not execute it from the IDE) is to reboot Windows.

That was heavy going now wasn't it? But do not worry if you do not understand it completely. If you stick to the example and do it like that you can't go wrong.

Once you have received the **WM_DESTROY** there is nothing you can do to keep the window alive. So what if we want to ask the user for confirmation first when he/ she closes the window?. In this case you have to process the **WM_CLOSE** message which you receive prior to **WM_DESTROY**. The example in the help file shows how to do that.

The only thing that is left for us to do now is to let Windows handle all messages that we did not handle. This is done by calling ***DefWindowProc*** in the default branch of the switch statement. For all the messages we have processed we return a value of zero.

```
      default:
           // We didn't process the message so let Windows do it
           return DefWindowProc(hWnd,msg,wParam,lParam);
    }
   return 0L;
}
```

5.5 Doing the painting

```
static void PaintAppWindow(HWND hWnd,HDC hdc,int iPaintCount,HBRUSH hEllipseBrush)
```

This function is declared as **static** since it will only be called within this module. It is passed the handle of the window, the handle of the display context, the count number of messages received so far, and a handle to a brush which we use to fill the ellipse. First we need to allocate the following local variables:

```
{ char   szText[50]; // Array holding the string displayed
  RECT   rcWnd;      // Dimensions of the Windows's client area
  HBRUSH hOrgBrush;  // Original Brush of display context
```

The first thing we do now, is to get the coordinates of the window's client area and store them in ***rcWnd*** which is a structure containing the left, top, bottom and right boundaries of the rectangle.

```
  GetClientRect(hWnd,&rcWnd);
```

Usually the upper left coordinate will be at 0/0 unless you change the windows origin. Hence the right and bottom values will contain the width and height of the client area.

We can now use this information to draw an ellipse touching each side of the window. Before we actually draw it though, we've got to select the fill brush we want to use into our display context.

All drawing primitives in Windows i.e. lines, rectangles, ellipses, arcs are always carried out with the pen and brush that is currently selected into the display context. The default is a black pen and a white brush. To change that you can call ***SelectObject*** and give it another pen, brush, color palette etc. ***SelectObject*** automatically detects the type of object you specified and replaces the current one. It also returns a handle to the object that was previously selected and this is a very important value to remember, since you are responsible to undo all changes before you release the display context again using ***EndPaint*** or ***ReleaseDC***. Failing to restore the display context is a major offence and will most likely be punished by the need to reboot your system. The effect does not occur immediately though but after hours of use. This is why these bugs are very difficult to find and require special tools to detect such as **BoundsChecker**.

All right, this is how we do it now:

```
  hOrgBrush=SelectObject(hdc,hEllipseBrush);                    // select the brush
  Ellipse(hdc,rcWnd.left,rcWnd.top,rcWnd.right,rcWnd.bottom);   // Draw the ellipse
  SelectObject(hdc,hOrgBrush);                                  // deselect the brush
```

Now all we need to do is write the text into the centre of the window. To do that we use ***wsprintf*** to generate a string consisting of the text and the number value of ***iPaintCount***. Then we set the background mode of the display context to transparent (otherwise it would be white) and draw the text using ***DrawText***.

```
  wsprintf(szText,"Paint Count: %d - Click inside window to repaint!",iPaintCount);
  SetBkMode(hdc,TRANSPARENT);   // Make text transparent
  DrawText(hdc,szText,lstrlen(szText),&rcWnd,DT_SINGLELINE|DT_CENTER|DT_VCENTER);
```

DrawText has several options that are all given with the last parameter and which can be combined using a binary or. See the description in the help file for details. Another option for outputting text is the function ***TextOut***.

6. Sort it out: A dialogue based Windows program

This Chapter describes a windows application based on a dialogue box which allows to enter a list of names and sort the list in ascending or descending order.

6.1 The Files

To compile and modify the program you need the following files:

DIALOG.C - the main code file

DIALOG.H - Header file

DIALOG.RC - the main resource file

`DIALOG.ICO` - Icon file containing program icon in binary form

`DIALOG16.EXE` - 16-bit executable file for Windows 3.x

`DIALOG32.EXE` - 32-bit executable file for Windows 95/ Windows NT

Example2.zip Click here to download the archive to your computer.

6.2 Designing the application

As I mentioned before, with this approach writing code only comes second. What you start with here, once you've created the project in the IDE is to call the ResourceWorkschop (Application Studio for the Microsofties) by double clicking on the .RC file. The RC files is created automatically by the resource editor, but since it is a plain ASCII file you can modify afterwards like any other text file. In this case we want to create a menu and a dialogue template for our main window. Most dialogues do not require menus, but I've implemented one here just to show you how it can be done.

In order to design our application window now, we first create a new resource of type Dialog (oooh, they speak American English...). You will be provided with an dialogue window that is empty except for the three default buttons OK, Cancel and Help that you will find in most dialogues. As we don't need them here they can be deleted. Now you add controls to this dialogue and place and size them as you want. There are five types of standard controls that you can use in Windows 3.1 and some more if you've got a resource editor for Windows95. Each control also has properties with determine the style and behaviour of the control. To access the properties double click on a control. If you want to find our more about a particular option in the property dialogue press the help button (I keep on saying that because most people don't use help files very much). At runtime i.e. during program execution you can send messages to controls in order to set them up or check their state and you will receive so called "***Notification Messages***" if the user changed data or the state of a control.

Here's a brief description for the six standard controls:

- **STATIC** This is normally text but it can also be an icon or a frame. Static controls are the simplest form of controls and there is nothing they do at runtime (that's why they're called static). Hence there are no special message that you can send to this control and you won't receive messages from them either.
- **EDIT** this allows the user to input text and numbers. Most edit controls are single line, but you can have a multi line edit control as well. To give you and idea how powerful edit controls are, have a look at NOTEPAD.EXE the Windows 3.1 text editor. This application is basically a window with a single edit control inside it. Edit controls accept messages beginning with **EM_** which allow you e.g. to select text or to retrieve the length of a line and they will notify you e.g. if text was changes by notification messages beginning with **EN_**.
- **LISTBOX** this type of control provides a single column list that you can fill with text items. In the properties you can set whether the list shows the items unordered or alphabetically ordered. In our case we go for the unordered option since we sort the list ourselves (this is a useful exercise in order to learn how to deal with list boxes). You can add as many items as you want and the ist box will automatically display a scrollbar when not all items fit on the screen. List boxes accept a whole range of messages all beginning with **LB_** and notify you if the user e.g. selects an item. Notification messages all begin with **LBN_**.
- **COMBOBOX** This is a combination of an edit control and a list box. Combo boxes are useful when a user can select on option out of a list like e.g. the current font. Most combo boxes you'll see in applications are of type drop-down which means, that the list box only pops up, when the user presses the little arrow down button which is placed behind the edit portion of the control. For most list box messages there is an equivalent message for combo boxes all beginning with **CB_**. Combo boxes will also notify you of changes made by the user by sending a **CBN_** notification.
- **BUTTON** probably the most often used control type in dialogue boxes. Buttons can be either push buttons (OK and Cancel are push buttons), check boxes (the ones with the little cross inside them when they are checked), radio buttons (make only sense when there is more than one of which only one can be selected) or group frames. The latter is the odd one out, since it is not a button at all. Don't ask me why they did it that way. There are a few messages and notification message which you won't need because there are functions in the Windows API which do the job.
- **SCROLLBAR** this is a control that provides a slide and two arrows at both ends to set a particular value. It is normally used on the right and bottom of a window to allow the user to move though the client area. However you can use scrollbars everywhere and for other purposes as well. Scrollbars do not accept messages and also have a different notification mechanism. When changed, vertical scrollbars send a **WM_VSCROLL** message to the parent window (not through the **WM_COMMAND**), horizontal scrollbars send **WM_HSCROLL**. The client i.e. you, is responsible for setting the scroll range and scroll position of a scrollbar using ***SetScrollRange*** and ***SetScrollPos***.

Now you've got the basic information you need and you can design something fancy. In all my years of Windows programming experience the design of dialogue boxes is still the most difficult and time consuming task. It sometimes takes me up to a couple of days to make a dialogue box look nice and intuitive to use. But if you are less of a perfectionist you can of course just throw the controls into the dialogue window. However it is not a very good idea to do it quick now, and perfectionise it later, because if you change the design after you've done the coding you'll face a lot more work than doing the design properly in the first place. The position and size of controls is not a problem since you can always change that without changing your code, but if you later on decide you need radio buttons instead of check boxes or you need to turn this list box into a combo box say, your up shit creek.

Another issue is the creation of control identifiers. In order to send and receive message from controls you need to give each of them a unique identifier which is simply an integer number. Both ResourceWorkshop and Application Studio help you by creating identifiers automatically although I prefer doing it myself. It's a bit more work, but you know exactly what's going on. Let me give you an example on this can be done:

Suppose you've got a dialogue which looks somewhat like this:

Then the resource script created by the ResourceWorkshop for this dialogue looks like this:

```
DIALOG_1 DIALOG 64, 99, 207, 46
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Authorisation"
FONT 8, "MS Sans Serif"
{
 CONTROL "Enter your name:", -1, "STATIC", SS_RIGHT | WS_CHILD | WS_VISIBLE | WS_GROUP, -1, 18, 60, 8
 CONTROL "", IDC_EDIT1, "EDIT", ES_LEFT | WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 64, 16, 79, 12
 CONTROL "&OK", IDOK, "BUTTON", BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 148, 6, 50, 14
 CONTROL "&Cancel", IDCANCEL, "BUTTON", BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 148, 24, 50, 14
}
```

The first line gives the dialogue name **DIALOG_1** followed by the type of resource which is **DIALOG** and the position and size of it. Now change **DIALOG_1** into something more meaningful like **DLG_AUTHORISE** and add a define statement in a header file which you include in both your code and the resource file. For example call the header file **DIALOG.H** and the follwoing line:

```
#define DLG_AUTHORISE      100
```

Then do the same thing with the identifier for the edit control which is set to **IDC_EDIT1**. If you change that to **IDC_NAME** say, you have to add a define for that as well in the header file like e.g.

```
#define IDC_NAME           100
```

Note that both values can be 100 since the first one is a resource and the second one only a identifier within this resource. By the way, **IDC_** stands for IDentifier Control as you might have guessed. But you don't need to stick to it. I normally start with the name of the dialogue the control belongs to so I would call it **DLGAUT_NAME** (for DiaLoGAUTorise_NAME). You do not need control identifiers for the OK and Cancel buttons since **IDOK** and **IDCANCEL** are predefined constants in the windows.h file and so is **IDHELP**. And you can give all your static controls the value of -1 unless you want to set the text of the control at runtime.

Another thing you might want to do in the end is to put your controls in order. This can either be done in the resource editor or by editing the .RC file directly. The order is important for two things:

- The first control in the definition which is not a static control will receive the input focus e.g. in the case above, the edit control would be active and you can enter text without clicking onto the edit control first. If the OK button would be first, then this would have the input focus and to enter text you'd have to click it or use the tab key.
- It determines in which order the control statements are stepped though when you use the tab key. This is important for all those who'd rather use the keyboard than the mouse and hence it should reflect the logical order.

All right, that was the visual part, now lets go and implement the code for it.

6.3 The procedure WinMain

The procedure ***WinMain*** is the entry point of the application and it normally only features a few lines of code to create window classes and create the main application window. In this example the ***WinMain*** features as many as three lines of code. If you want more information about the ***WinMain*** and the parameters read the section about the procedure ***WinMain*** in the previous chapter.

```
int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpCmdLine, int nCmdShow)
```

The first parameter **hInst** is the handle to the program instance which we will need later on for various other functions. Since it is constant and valid for the whole lifetime of this instance of the program we store it in the global variable **hInstance**.

```
hInstance=hInst;
```

Now we create our application window which is a dialogue box in this case. The function will not return before the dialogue is closed. This type of dialogue is called a "modal" dialogue. There is another type of dialogue called "modeless" which will not wait till the dialogue is closed and return immediately. If you want to find out more about those see the windows SDK help file for ***CreateDialog***.

```
DialogBox(hInstance,MAKEINTRESOURCE(DLG_MAIN),NULL,MainDlgProc);
```

In its second parameter ***DialogBox*** requires the name of the template that is used for the dialogue. This is the one we defined earlier with the **ResourceWorkshop**. You can here use either a string or better use a numeric identifier. In the latter case you have to use ***MAKEINTRESOURCE*** to pass the value to the function. I used **DLG_MAIN** as a numeric identifier which I defined in the dialog.h header file. This file must then also be included by the resource file (***dialog.rc***).

Sometimes it can happen, that you call dialogue box, but nothing happens and ***DialogBox*** returns immediately with a return value of zero. In most cases this is due to an invalid dialogue template identifier. Check first, whether both the .C file and the .RC file include the header file where

the numeric identifier is defined and that it is also used as the template name in the .RC file. If this is correct and there is still no dialogue box, check the dialogue box procedure (remember to return **FALSE** if you do not process a message) and if you use custom controls, whether all window classes are registered.

The third parameter of ***DialogBox*** is the handle to the parent window. In this case we do not have one, that's why we pass it **NULL**. The last parameter is a pointer to the dialogue box procedure that will handle all events. In the early days of windows this procedure address could not be given to the function directly and you had to a function called ***MakeProcInstance*** before you called ***DialogBox*** and ***FreeProcInstance*** afterwards. This was a major pain especially for modeless dialogues but luckily all modern windows compilers automatically deal with this problem. Under Win32 this is obsolete anyway and so there is no reason to bother you with the reason for that.

Well that was it really. If the dialogue is closed, we terminate by returning **TRUE**. It's as easy as that!

```
return TRUE;
```

6.4 The dialogue box procedure

The main difference between the previous approach described in the previous chapter and this one is, that the main window of the application is based on a dialogue box rather than a custom window. Of cause a dialogue box is also a window but the difference is, that the window procedure for a dialogue window is defined in Windows's itself. This window procedure will then call a dialogue procedure which we have to provide in our program in order to handle events.

A dialogue procedure gives you exactly the same parameters as a window procedure but the return value is here of type **BOOL** rather than of type **LONG**. Whereas in a window procedure we return zero (0L), if we have processed a message and call ***DefWindowProc*** if we didn't. In a dialogue procedure we return **TRUE** if we processed it and **FALSE** otherwise. Here is a general outline for a dialogue procedure:

```
BOOL FAR PASCAL DlgProc(HWND hDlg,UINT msg,WPARAM wParam,LPARAM lParam)
{
  switch (msg)
   {
     case WM_...
          /* process a message */
          break;
     default:
          /* let Windows handle the message */
          return FALSE;
   }
  return TRUE;
}
```

The most important messages to process for a dialogue procured are **WM_INITDIALOG**, **WM_COMMAND** and may be **WM_DESTROY**. **WM_INITDIALOG** is sent to the dialogue procedure instead of a **WM_CREATE** message (which is only sent to window procedures). It can be used to initialise dialogue controls and allocate required resources. Use **WM_DESTROY** to clean up afterwards if necessary. In a window procedure one of the most important messages to handle is the **WM_PAINT** message and you might wonder where that is here. Well, of course you get this message for a dialogue window well, but normally there is no reason to handle it.

6.4.1 The WM_INITDIALOG message

Let's now look at communication with controls and the initialisation of the dialogue with the **WM_INITDIALOG** message. In my example a good thing to start with, is to disable all controls that are useless at the beginning. These are the buttons for adding a name to the list and sorting the list. Note: All controls are enabled by default unless you specify otherwise in the dialogue template.

```
case WM_INITDIALOG:
     EnableWindow(GetDlgItem(hDlg,IDC_ADDNAME),FALSE);
     EnableWindow(GetDlgItem(hDlg,IDC_SORTLIST),FALSE);
```

Since ***EnableWindow*** requires a window handle we have to get the handle of the control first by calling ***GetDlgItem***, which we give a handle to the dialogue window and the numeric control identifier. Set the second parameter of ***EnableWindow*** to **TRUE** if you want to enable the control or to **FALSE** to disable it. Disabled controls are displayed with grey text and are unavailable to the user. You can enable or disable all types of controls just like any other window.

Next we set the default options for the radio and check buttons. ***CheckRadioButton*** takes the identifier of the first and last radio button in the group and the one you want to set the check mark to. It is important that the identifiers for radio buttons all have consecutive numbers assigned to them i.e. if the numeric identifier for the first button is defined with a value of 100 then the control identifiers for the other radio buttons of that group have to be 101, 102, 103 and so on. ***CheckDlgButton*** is used for the check box.

```
     CheckRadioButton(hDlg,IDC_ASCENDING,IDC_DESCENDING,IDC_ASCENDING);
     CheckDlgButton(hDlg,IDC_CASEINSENSITIVE,TRUE);
```

And last not least, we set the maximum length of text the user can type into the edit control. There is no API function for that, but we can send a message and the message for this kind of thing is **EM_LIMITTEXT**. The maximum is then **MAXNAMELEN** which I've defined earlier on as 50 characters.

```
      SendDlgItemMessage(hDlg,IDC_NAME,EM_LIMITTEXT,MAXNAMELEN,0L);
      break;
```

Now that we've got that sorted, we can go on the **WM_COMMAND** message which.

6.4.1 the WM_COMMAND message

The **WM_COMMAND** is the heart of a dialogue box procedure since it handles both menu and control events. Unfortunately there is a slight difference here between Win16 and Win32. In Win16 where ***wParam*** is 16 bit and ***lParam*** 32 bit wide, ***wParam*** contains the identifier of the control and ***lParam*** contains both the handle of the control window (in the low order word) and a notification code (in the high order word). In Win32 ***wParam*** and ***lParam*** are 32 bit, ***wParam*** and contains both the control identifier (in the low order word) and a notification code (in the high order word) and ***lParam*** contains the handle to the control window only. Are you with me? Don't worry, here is how to keep everything nice, easy and independent:

First we define two macros, one for each API version:

```
#ifdef WIN32
  // Win32
  #define CTLID   LOWORD(wParam)       // Control ID for WM_COMMAND
  #define CTLMSG  HIWORD(wParam)       // Notification Message of Control
  #define HCTL   (HWND)lParam          // window handle of control

#else
  // Win16
  #define CTLID   wParam               // Control ID for WM_COMMAND
  #define CTLMSG  HIWORD(lParam)       // Notification Message of Control
  #define HCTL   (HWND)LOWORD(lParam)  // window handle of control

#endif
```

It is best to put that in your main header file, so it is available in every source code module. Then you can use it by processing the **WM_COMMAND** message in the following way:

```
  case WM_COMMAND:
       switch(CTLID)     // determine the control the message came from
        {
          case IDC_NAME: // Notification message from edit control
               if (CTLMSG==EN_UPDATE)
                 {
                   ..... // something changed in the edit field
                 }
               break;
          case IDC_....  // add case statements for other controls
               ......    // and some code to handle the event
               break;
        }
       break;           // end of WM_COMMAND
```

The first thing I am doing in my example, is to check whether there is any text at all in the edit window called **IDC_NAME** and enable or disable the "Add name to list" push button accordingly. So we say:

```
          case IDC_NAME:
              if (CTLMSG==EN_UPDATE)
                { int len=SendDlgItemMessage(hDlg,IDC_NAME,EM_LINELENGTH,0,0L);
                  EnableWindow(GetDlgItem(hDlg,IDC_ADDNAME),len);
                }
              break;
```

This reads in natural language as follows: If text has been altered in the edit window [***if (CTLMSG==EN_UPDATE)***] then get the number of characters in the edit control [***len=SendDlgItemMessage(hDlg,IDC_NAME,EM_LINELENGTH,0,0L)***] and enable the button **IDC_ADDNAME** according to number of characters [***EnableWindow(GetDlgItem(hDlg,IDC_ADDNAME),len)***]. If you've got that, then you've understood the basic principle and the rest is just a variation.

The next thing the user is likely to do is to press the "Add name to list"-button or press Return, which is the same thing as this is our default push button. Normally the OK button is the default push button but we haven't go one here. You can make any button your default push button and in

the resource editor you can specify which on it should be. For buttons we do not need to check the notification message since buttons have only one for the event that the button is pressed and hence **CTLMSG** will always be zero. What we need to do in this case now, is to get the text in the edit control and add it to the list of names. Therefore we allocate a buffer for the text first and fill it by calling ***GetDlgItemText***, giving that the control identifier of the edit control, a pointer to the text buffer and the maximum number of characters we want to read.

```
 case IDC_ADDNAME:  // Button Add name was clicked
     { char szName[MAXNAMELEN];
       GetDlgItemText(hDlg,IDC_NAME,szName,MAXNAMELEN);
```

To add the text to the list, we send the message **LB_ADDSTRING** to the list control **IDC_NAMELIST**. To avoid compiler warnings we cast ***szName*** to the correct type.

```
SendDlgItemMessage(hDlg,IDC_NAMELIST,LB_ADDSTRING,0,(LPARAM)(LPSTR)szName);
```

If you want to insert the name at a particular position then you can use LB_INSERTSTRING instead, giving it the list index where you want to insert it in the fourth parameter. Again there is lots of information about this in the SDK help file.

To make our program convenient to use, we clear the text in the edit control and set the input focus to it again so the user can type in another name immediately.

```
       SetDlgItemText(hDlg,IDC_NAME,NULL);
       SetFocus(GetDlgItem(hDlg,IDC_NAME));
```

Done that all that is left to do for this event is to enable the "Sort" button since there are now strings in the list box.

```
     EnableWindow(GetDlgItem(hDlg,IDC_SORTLIST),TRUE);
   }
   break; // end of IDC_ADDNAME
```

As it does not make sense sorting the list with only one item in, this could actually be improved by checking how many list items are in the list already. and enabling the button only if there is more than one. But you can add that easily yourself by sending a **LB_GETCOUNT** message to the list box and evaluating the return value.

The user can now type in names and add them to the list box. The next thing we've got to consider is the user pressing the "Sort" button to sort the list. This task is performed by the procedure ***SortList*** which is coded later on in the source file. But before we call this procedure we've got to get the sort options from the radio buttons and the checkbox control. Therefore we call ***IsDlgButtonChecked*** which will return **TRUE** if it is and **FALSE** if the button is not checked.

```
case IDC_SORTLIST:
    { BOOL CaseInsensitive=IsDlgButtonChecked(hDlg,IDC_CASEINSENSITIVE);
      BOOL bDescendingOrder=IsDlgButtonChecked(hDlg,IDC_DESCENDING);
      SortList(hDlg,IDC_NAMELIST,bCaseInsensitive,bDescendingOrder);
    }
    break;
```

Now our application is fully operational. However there is one little serious problem left: the user won't be able to close the dialogue which will in this case also terminate the program. For that we have got the Quit command in the menu and in dialogues you will also get a **IDCANCEL** event though the **WM_COMMAND** if the user presses the escape key or closes the window with the close command in the system menu (in Windows95 there is also a close button in the title bar).

Let's do one after the other and deal with the Quit command in the menu first. As with control notifications we receive a **WM_COMMAND** message when the user selected the menu command. In this case we close the dialogue by calling ***EndDialog***. The first parameter for this function is again the handle of the dialogue window and the second one is the return value for the calling function. You might have forgotten by now, but we called this dialogue in the ***WinMain*** with the ***DialogBox*** command. Since we do not evaluate the return value there we can return basically anything, but in an ordinary dialogue box with a OK and Cancel button you would return **TRUE** (or another positive value) if the user pressed OK and **FALSE** if he/ she cancelled. Anyway, this is what it looks like:

```
case IDM_QUIT:
     EndDialog(hDlg,TRUE);
     break;
```

After you've called ***EndDialog*** you will receive some more messages like the **WM_DESTROY** for example. Use this to free resources like memory that you have allocated if any.

To deal with the other ways to close the dialogue mentioned above you need to add some code for **IDCANCEL**. In this case I ask the user whether he/ she really wants to close the program with a ***MessageBox***. It will feature a little question mark icon (**MB_ICONQUESTION**) and a Yes and No button (**MB_YESNO**). If termination is confirmed then ***EndDialog*** is called.

```
case IDCANCEL:
    { int answer=MessageBox(hDlg,"Do you really want to quit?" ,"Confirm", MB_ICONQUESTION|MB_YESNO);
         if (answer==IDYES) EndDialog(hDlg,FALSE);
    }
    break;
```

6.5 Sorting the list

Finally let's have a brief look at the procedure used for sorting the list which uses a simple bubble sort algorithm. Again this is just to demonstrate how to communicate with controls and it is not a very efficient way of doing it. And after all in most cases there is not need to sort list boxes because you get the sort option for free if you specify this property for the list box.

Here's the procedure in which we first allocate a number of integers and two buffers to store both strings that we want to compare

```
void SortList(HWND hDlg,int idList,BOOL bCaseInsensitive,BOOL bDescendingOrder)
{ int  nItems,i,j;
  BOOL bSwap;
  char buffer1[MAXNAMELEN],buffer2[MAXNAMELEN];
```

Now we need to know how many items are in the list. Therefore we send a message **LB_GETCOUNT** message to the list box which will return the number of list items.

```
   nItems=SendDlgItemMessage(hDlg,idList,LB_GETCOUNT,0,0L);
```

In this case it is no problem if there are no entries in the list i.e. ***nItems*** is zero, since the following two for loops will handle it correctly. But how about other cases where you need a minimum of on list entry say. Well, normally it should not be necessary to check for that here. In this program for example we can be sure that there is at least one item in the list box otherwise the "Sort List" push button is disabled. This is an important matter! It is always better to prevent the user from making invalid input than to tell him/ her later with an error message that the input was invalid. And with windows you have the opportunity to do that easily. Take the editing of names as another example. Theoretically one could input an empty string to the list box, but since we only enable the "Add name to list"-button when at least one character is in the edit field, this can never happen. This rule applies not just to dialogue boxes. Menu command and toolbar buttons can and should be treated in the same manner.

Now we enter the first loop and copy the text of the first list item into our buffer. Again we send a message to the list box to do that. If you're not sure how long the string is and whether your buffer is big enough, you can send **LB_GETTEXTLEN** first and allocate sufficient space.

```
   for (i=nItems;i>0;i--)
    {
     SendDlgItemMessage(hDlg,idList,LB_GETTEXT,0,(LPARAM)(LPSTR)buffer1);
```

In the inner loop we get the next string in the list, compare the strings and if neccessary we swap them.

```
     for (j=1;j<i;j++)
        { // Get the text of the next item in the list
          SendDlgItemMessage(hDlg,idList,LB_GETTEXT,j,(LPARAM)(LPSTR)buffer2);
          // compare the two strings
          if (bCaseInsensitive) bSwap=(lstrcmpi(buffer1,buffer2)>0);
                           else bSwap=(strcmp(buffer1,buffer2)>0);
          if (bDescendingOrder) bSwap=!bSwap;
          // Swap the items if necessary
          if (bSwap)
            { // swap the strings
              SendDlgItemMessage(hDlg,idList,LB_DELETESTRING,j,0L);
              SendDlgItemMessage(hDlg,idList,LB_INSERTSTRING,j-1,(LPARAM)(LPSTR)buffer2);
            }
          else lstrcpy(buffer1,buffer2);
        }
    }
}
```

Unfortunately there is no message to set the text of a list box item. But what we can do is delete the current string by sending **LB_DELETESTRING** and then insert it again at the previous position with **LB_INSERTSTRING**.

7. How do I do...

Now this is the stage, where you should be able to stand on your own two feet, however weak they might still be. What you need now, is some guidelines on where to go, or better how to be able to go where you want to go. In this chapter I will give you some clues how to do certain things that are useful for many applications.

You don't have to read this chapter in the given order. Instead you can look up a topic you want further information about and try to implement it into one of the example programs.

Common Dialogues

Common Dialogues are a very important feature introduced with Windows 3.1 in order to standardise and simplify actions required most applications for both the users and the programmer. When Windows 3.0 started to get a really big success it soon became a problem that every application used a different methodology for similar things. The File Open dialogue is probably the best example for that. First it was not to easy to program although you'd need it in almost every program, and second users didn't find it too exciting facing a different interface every time. So Microsoft came up with the so called Common Dialogues which provide a consistent user interface for the most common tasks for every application. You can of course still do it all yourself, but you will find it a lot more convenient and easy to use them. The common dialogues reside in the dynamic-link library **COMMDLG.DLL** which provides the following functions:

ChooseColor Opens a dialogue in which the user can create and select a colour. You will find this dialogue coming up e.g. if you choose to customise your desktop colours in the control panel.

ChooseFont Allows to select a font and its type and size and colour properties. Optionally you get a preview of what a particular text with this font would look like.

FindText This is a modeless dialogue for searching text in a document. This will of course like all other common dialogues only provide a dialogue box not the actual search in your document.

ReplaceText The same as above but with the additional option to replace a particular text.

GetOpenFileName This is the nicest, quickest and most user friendly way to create a dialogue box that allows the user to select a file to open. There are many options and you can even use your own dialogue template, but the functionality of the dialougue like diplaying and changing directories and drives will be completely handled for you.

GetSaveFileName Ditto but for saving files.

PrintDlg A dialogue to set up printer properties

In order to common dialogues you first need to include the header file **COMMDLG.H** in your code file. You then call one of the above functions with the parameters described in the SDK help file. If you find the description given in there a bit too technical, here is a little example how to display a file open and file save dialogue.

First define the maximum length of the filename in your main header file:

```
#define MAXFILENAMELEN 80
```

Then define following function in one of your source modules. Change the member ***Flags*** as appropriate.

```
BOOL DlgGetFileName(HWND hWnd,LPSTR lpszTitle,LPSTR lpszFormat,LPSTR lpszFileName,BOOL save)
{  OPENFILENAME   of;
   int            result;
   // Initialize the OPENFILENAME members
   of.lStructSize       = sizeof(OPENFILENAME);
   of.Flags             = OFN_HIDEREADONLY | OFN_NOCHANGEDIR | OFN_SHOWHELP;
   of.hwndOwner         = hWnd;
   of.hInstance         = hInstance;
   of.lpstrFile         = lpszFileName;
   of.lpstrFilter       = lpszFormat;
   of.nMaxFile          = MAXFILENAMELEN;
   of.lpstrInitialDir   = NULL;
   of.lpstrTitle        = lpszTitle;
   of.lpTemplateName    = 0;
   of.lpfnHook          = NULL;
   // Display the dialog
   if (save) result=GetSaveFileName(&of);
        else result=GetOpenFileName(&of);
   return result;
}
```

In the module containing the window procedure of your main window you should then define a string containing the name and extension of your file format and allocate a buffer for the filename:

```
static const char szFileFormat[]={ "My file\0*.MYF\0" };
static char szFileName[MAXFILENAMELEN];
```

And finally you need to call the function in your window or dialogue procedure with somewhat like:

```
case WM_COMMAND:


     switch(wParam)
       {
        case IDM_OPEN:
             if (DlgGetFileName(hWnd,"Open a file",szFileFormat,szFileName,FALSE))
               { int hfile;
                 hfile=_lopen(szFileName,OF_READ);
                 ...
                 ...
               }
             break;
        case IDM_SAVE:
             if (DlgGetFileName(hWnd,"Save a file",szFileFormat,szFileName,TRUE))
               { int hfile;
                 hfile=_lcreat(szFileName,0);
                 ...
                 ...
               }
             break;
```

If you need to extend the functionality of your dialogue e.g. in order to offer a preview of the file you can do that by providing your own dialogue template and/ or specifying a hook procedure. Find information about this in the SDK help file.

Setting control colours and using 3D controls

When you implement your own dialogues you will find, that they still look somewhat different from all those fancy ones you see in other programs. What is missing is just a nice grey background and a nice 3D look of all the controls. Surely your application doesn't gain any more functionality with 3D of radio buttons, check boxes, edit controls and the lot, but if you want users to accept your program you'd better go for the fancy ones. Windows 95 already automatically provides a 3D look for controls, all that is missing here is a nice grey background color (the default as you can see is white).

Now if it just the background color that bothers you, then all you have to do is to process the ***WM_CTLCOLOR*** message. This message is sent from the control to the parent window (which is normally a dialogue) to allow different colours to be used. Unfortunately there is another difference between Win16 and Win32. Note, that it does not matter on which version of Windows you run the program only which you compile it for. In Win16 you only get the one message mentioned above and the lower word of ***lParam*** indicates what type of control sent the message. This can be a button, an edit control, a list box, a combo box, a static control or the dialogue itself. In Win32 there are six different messages, one for each control type.

This is how you'd do it for Win16:

```
case WM_CTLCOLOR:
     switch(HIWORD(lParam))
      {
         case CTLCOLOR_DLG:
         case CTLCOLOR_STATIC:
         case CTLCOLOR_BTN:
              SetBkColor((HDC)wParam,RGB(192,192,192));
              return GetStockObject(LTGRAY_BRUSH);
      }
     return FALSE; // use the defaults
```

and the equivialent for Win32:

```
case WM_CTLCOLORDLG:
case WM_CTLCOLORSTATIC:
case WM_CTLCOLORBTN:
     SetBkColor((HDC)wParam,RGB(192,192,192));
     return GetStockObject(LTGRAY_BRUSH);
```

In both cases you set the background color for text in the display context given in ***wParam*** using ***SetBkColor*** and return a handle to a brush. In this case I have obtained the handle to one of the predefined brushes from a call to ***GetStockObject***. This function can also be used for pens and fonts as you can see when looking it up in the SDK help file. However if you want another color you have to create a brush using

CreateSolidBrush which you best put in the **WM_INITDIALOG**. The handle you're receiving must be static declared as static and you can then return it on any call to **WM_CTLCOLOR**. Don't forget to destroy it in the **WM_DESTROY** using ***DeleteObject***.

A disadvantage of this approach is, that you have to include it in every dialogue procedure and you won't get the nice 3D effect unless you're using Windows 95. Therefore you might rather go for using Microsoft's 3D control library. This library is called **CTL3DV2.DLL** and you'll probably have it already in your Windows system directory. If not, then you should have a look on your compiler CD or any other since it is installed by most programs and can be distributed royalty free. In order to use it, you have to add the file **CTL3DV2.LIB** to your project just like if you'd add another source code module. This will make the DLL load automatically when your program is started.

All you've got to do now, is to include the **CTL3D.H** file

```
#include <clt3d.h>
```

and add the following lines to the ***WinMain***

```
   Ctl3dRegister(hInstance);
   Ctl3dAutoSubclass(hInstance);
```

This should be done prior to anything else. You then enter your dialogue box or message loop as usual and afterwards, just before you leave the ***WinMain*** you call

```
   Ctl3dUnregister(hInstance);
```

Now all your dialogues will have a grey background, nice looking dialogue boxes and 3D controls. If you are interested in more controls like tree views, multi column list boxes, spin controls etc. you might want to look at the **COMMCTRL.DLL**. Information about it can be found on the Microsoft Developer Network Library.

Customizing Dialogue Controls

In the dialogue example I have shown you how to use standard controls to input and output data. Standard controls like buttons and list boxes are handy because they make programming quick and easy. But how capable and flexible are they?

First of all, you can use standard controls not just in dialogue boxes but also in your "normal" windows. This for example creates a button inside the window specified in ***hWnd***:

```
CreateWindow("BUTTON","&Delete",BS_PUSHBUTTON | WS_CHILD | WS_VISIBLE, 10, 10, 50, 20, hWnd,
                       IDC_DELETE, hInstance, NULL);
```

Fair enough, but how about displaying other type of data like graphics, images or both mixed with text. And how about if you need a control with a different behaviour?

There are three techniques which all allow you to extend the functionality of your dialogues and controls:

- Using "Owner Draw" controls. Buttons, list boxes and combo boxes have a property called "owner draw" which you can set, if you wish to perform the output yourself. This is helpful, if the nature of the control you need matches one of those controls and you only want a different representation. This allows you e.g. to create buttons with images on, list boxes with different colour list items or combo boxes with items consisting of an icon and a text string.
- "Subclassing" a control. This technique that can be used with any window intercepts the window procedure and allows you to filter window messages. This is useful if the extension to an existing control/ window your aiming for is not in the display but rather the type of interaction. So you can e.g. use subclassing to allow drag and drop of elements from and to a list box.
- Creating your own control. This is useful if you want to do something completely different like e.g. a dice control. Here you've got to do all yourself and you can define your own style properties, messages and notification messages. To create your own control you have to register a window class first and define the window procedure for this class. Then simply create a window derived from this class. You can also use your own classes in dialogue templates of course.

Owner draw controls

Owner draw controls send you a **WM_DRAWITEM** message. The ***lParam*** of this messages contains a pointer to a **DRAWITEMSTRUCT** structure that contains the display context, the rectangle of the item in the display context and a pointer to the item data. Here is an example for an owner draw list box that displays a list of icons:

First create a list box inside your dialogue end set its properties of "owner draw" and "has strings". The resource statement of the list box in the .RC file should looks something like:

```
CONTROL "", IDC_ICONLIST, "LISTBOX", LBS_NOTIFY | LBS_OWNERDRAWFIXED | LBS_HASSTRINGS | WS_CHILD |
```

```
                WS_VISIBLE | WS_BORDER | WS_VSCROLL, 30, 127, 111, 55
```

Icons have a width and height of 32 pixels. In the **WM_INITDIALOG** message we first set the height of the list box items to 32:

```
SendDlgItemMessage(hDlg,IDC_ICONLIST,LB_SETITEMHEIGHT,0,MAKELPARAM(32,0));
```

Now you can add the items. Each requires text that we add with **LB_ADDSTRING** and and icon handle. This handle is set by sending a **LB_SETITEMDATA** message.

```
i=SendDlgItemMessage(hDlg,IDC_ICONLIST,LB_ADDSTRING,0,(LPARAM)(LPSTR)"myIcon");
SendDlgItemMessage(hDlg,IDC_ICONLIST,LB_SETITEMDATA,i,(LPARAM)hIcon));
```

Now you have to handle the **WM_DRAWITEM** message in the dialogue procedure:

```
case WM_DRAWITEM:
     { LPDRAWITEMSTRUCT lpdis=(LPDRAWITEMSTRUCT)lParam;
       HBRUSH  hOrgBrush;
       int     iBkColor,iTxtColor;
       RECT    rcText;
       char    szText[50];
       if (((int)lpdis->itemID)<0) break;
       // Determine the colors
       if (lpdis->itemState & ODS_SELECTED)
         { iBkColor=COLOR_HIGHLIGHT;iTxtColor=COLOR_HIGHLIGHTTEXT; }
       else
         { iBkColor=COLOR_WINDOW;iTxtColor=COLOR_WINDOWTEXT; }
       // Erase the background
       hOrgBrush=SelectObject(lpdis->hDC,CreateSolidBrush(GetSysColor(iBkColor)));
       PatBlt(lpdis->hDC,1,lpdis->rcItem.top,lpdis->rcItem.right-2,lpdis->rcItem.bottom-lpdis->rcItem.top,PATCOPY);
       DeleteObject(SelectObject(lpdis->hDC,hOrgBrush));
       // Draw the icon
       DrawIcon(lpdis->hDC,1,lpdis->rcItem.top,(HICON)lpdis->itemData);
       // Draw the text
       rcText=lpdis->rcItem;
       rcText.left=36;
       SendMessage(lpdis->hwndItem,LB_GETTEXT,lpdis->itemID,(LPARAM)(LPSTR)szText);
       SetBkMode(lpdis->hDC,TRANSPARENT);
       SetTextColor(lpdis->hDC,GetSysColor(iTxtColor));
       DrawText(lpdis->hDC,szText,lstrlen(szText),&rcText,DT_SINGLELINE|DT_VCENTER);
       // Draw the Focus rectangle
       if (lpdis->itemState & ODS_FOCUS)
           DrawFocusRect(lpdis->hDC,&lpdis->rcItem);
     }
     break;
```

Subclassing a control

To understand what subclassing can do, consider a list box that outputs a list of tasks that can only be processed in a sequential order. You indicate the current task by highlighting the corresponding list item (using **LB_SETCURSEL**). What you now don't want is that the user can select any other item. To prevent the user from doing this, you can disable the list box, but unfortunately this will also disable the list box's scrollbar and all items are greyed.

A solution for this is to subclass the window procedure for the list box and filter out all mouse and keyboard messages. All other messages must be processed as normal so we need to pass them on to the original list box procedure.

```
static FARPROC lpOrgListboxProc;

LRESULT FAR PASCAL ListSubclassProc(HWND hWnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
  switch(msg)
   {
     case WM_LBUTTONDOWN:
     case WM_MOUSEMOVE:
     case WM_LBUTTONUP:
     case WM_CHAR:
          // do nothing
          break;
     default:
          // process as normal
```

```
          return CallWindowProc(lpOrgListboxProc,hWnd,msg,wParam,lParam);
   }
return 0L;
}
```

Now you can subclass the control in the **WM_INITDIALOG** procedure by storing the original window procedure in ***lpOrgListboxProc*** and setting the window procedure for the list box control to your own procedure.

```
lpOrgListboxProc=(FARPROC)GetWindowLong(GetDlgItem(hWnd,IDC_TASKLIST),GWL_WNDPROC);
SetWindowLong(GetDlgItem(hWnd,IDC_TASKLIST),GWL_WNDPROC,ListSubclassProc);
```

Bitmaps

Bitmaps are images consisting of n times m pixels (picture elements). You can create and modify bitmaps (stored in .BMP files, which is the standard bitmap format for Windows) with Paintbrush, MS Paint or any other pixel orientated drawing program. The easiest way to use a bitmap image in your application is to include it as a resource. The resource file code for a bitmap is:

```
ID_IMAGE BITMAP "myimage.bmp"
```

Where **ID_IMAGE** it the name or numeric identifier of the bitmap and **"myimage.bmp"** is the name of the file containing the bitmap.You can then load it in your program with the command ***LoadBitmap*** which gives you a handle to the bitmap (type **HBITMAP**). e.g.

```
hImage=LoadBitmap(hInstance,MAKEINTRESOURCE(ID_IMAGE));
```

The only problem you have now, is to bring the bitmap on the screen. This requires a little bit of preparation. First you need a memory display context. Let me just briefly explain what that is. A memory display context is basically the same as a window but instead of mapping output to the screen all drawing goes to memory. Where the output area of a "normal" display context is the client area of a window which is constrained by the window size and the colour depth (bits required to display the maximum number of colours) of the current display mode, the output area of a memory display context is constrained by the width, height and colour depth of the bitmap selected into it. Therefore a memory display context needs a bitmap just as a window needs the screen.

The reason why we need a memory display context is that we cannot select a bitmap directly into a window display context, but we can copy data from one display context into another, provided that they are compatible. The following code creates a memory display context that is compatible with the display context of the screen:

```
hMemDC=CreateCompatibleDC(NULL);
```

The value of **NULL** always creates a display context that is compatible with the screen. Optionally you can give it the handle of your window's display context, but it won't make any difference. If you use bitmaps more than once I recommend that you create a memory display context at the beginning in the ***WinMain*** and make the handle global. It is always useful to have a memory display context around.

Next let's see what we can do with that. Suppose your bitmap is 300 times 200 pixels big and you want to display it at position 0,0 of your window then your **WM_PAINT** message handling could look somewhat like this:

```
case WM_PAINT:
     { PAINTSTRUCT ps;
       HBITMAP hOrgBitmap;
       HDC hdc;
       hdc=BeginPaint(hWnd,&ps);
        hOrgBitmap=SelectObject(hdc,hImage);
        BitBlt(hdc,0,0,300,200,hMemDC,0,0,SRCCOPY);
        SelectObject(hdc,hOrgBitmap);
       EndPaint(hWnd,&ps);
     }
     break;
```

The function which does the job is ***BitBlt*** which stands for Bit Block Transfer. It copies an area from a source display context, which is the memory display context in this case, into a destination display context, which is our window in this case. This works also the other way round of course and you can also copy from one memory context into another or from window context into another. Before you use a ***BitBlt*** with a memory display context however, you have to select a bitmap into it, which you do with ***SelectObject***. As with all other GDI objects (pens, brushes, fonts and palettes) you have to remember the original object and restore the display context if you do not need it any more.

Instead of ***BitBlt*** you can also use ***StretchBlt*** which stretches or compresses the bitmap from its original size into the destination rectangle.

There is now only one thing to remember: Cleaning up afterwards. We have allocated two resources, a bitmap and a display context, which you have to delete before terminating the program. Therefore we use

```
DeleteObject(hImage);
```

to delete the bitmap and

```
DeleteDC(hMemDC);
```

to delete the memory display context.

This was just scratching the surface of bitmaps. There is so much to say and know about bitmaps that I could easily write another tutorial of this size just about it. The problems begin, when you start dealing with Device Independent Bitmaps (DIBs) and Color Palettes. But if you need to know more about that, you really need a book.

Printing

Have you ever tried to print out a nice graphic (with your own program of course) in a DOS or UNIX program? It not, then I can tell you that it takes ages (and a lot of reading in the printer manual) to get a reasonable result on your own printer and is virtually impossible to get it right for every printer. If you aim for **WYSIWYG** (What You See Is What You Get) or even just a proper print preview, you just have to forget it.

Is Windows any better than? Well it is, and not just a bit. Printing anything under Windows is really trivial and you can not just use almost any font, font size and font style you like but also print out all kind of graphics including bitmaps without any difficulties.

Suppose you've got a function called ***PaintAppWindow*** which you normally call in the **WM_PAINT** branch of your window procedure like:

```
case WM_PAINT:
     { PAINTSTRUCT ps;
       HDC hdc;
       hdc=BeginPaint(hWnd,&ps);
        PaintAppWindow(hdc);
       EndPaint(hWnd,&ps);
     }
     break;
```

No matter what this function does, you can have exactly the same output on your printer just by calling it with a printer display context instead of a screen display context. Hence the only thing you need to do is something like the following function, which reads the name, driver file name and port number of the current printer (the one you've specified as the standard printer in the control panel) and then creates a display context for that printer:

```
PrintWindow(void)
{ char sDevice[160];
  char sName[64],sDriver[80],sPort[16];
  HDC hPrinterDC;
  GetProfileString("WINDOWS", "DEVICE", "", sDevice,sizeof(sDevice));
  sscanf(sDevice, "%64[^,],%80[^,],%16[^,]", sName, sDriver, sPort);
  hPrinterDC=CreateDC(sDriver,sName,sPort,NULL);
PaintAppWindow(hPrinterDC);
  DeleteDC(hPrinterDC);
}
```

Not too bad, is it? OK, may be I have exaggerated a bit. This works, but in reality you'll have to do a bit more to get a WYSIWYG result. For example to get the equivalent font size or a font you're using for the screen you have to do the following:

```
int GetFontSize(HDC hPrinterDC,int nScreenHeight)
{ POINT pt;
  pt.x=0;
  pt.y=-MulDiv(nScreenHeight,GetDeviceCaps(hPrinterDC,LOGPIXELSY),72);
  DPtoLP(hPrinterDC,&pt,1);
  return pt.y;
}
```

Use the return value now to fill the ***lfHeight*** property of a **LOGFONT** structure and create a font for printing using ***CreateFontIndirect***, select it into your display context using ***SelectObject*** and off you go.

To retrieve the properties of the printer display context you can call ***GetDeviceCaps*** as in the example above. To determine the width and height of a printer page in pixels for example you can call ***GetDeviceCaps*** with the handle of the printer display context and either **HORZSIZE** or **VERTSIZE**.

Finally you may want to provide a printer configuration dialogue with which the user can set up the printer. Just make use of the common dialogues for that and call ***PrintDlg***.

Sound

To play sound you can easily make use of the Multimedia Control Interface (MCI). Information about this can be found in the Multimedia Reference help file (**WIN31MWH.HLP**). The easiest way is to call ***SndPlaySound*** to play back a .WAV file (digital sound). This is how you do it:

```
#include <mmsystem.h>
....
sndPlaySound("C:\\WINDOWS\\DING.WAV",SND_SYNC);
....
```

Midi files (extension .MID) are a bit more difficult to play back. Basically what you have to do is to call ***mciSendCommand*** to open a MCI device and then send commands to this device to play them back. There is lots more you can do with ***mciSendCommand*** like recording and playing back WAV files as well.

Getting more information

Windows programming is a never-ending field of learning you'll probably never be able to know everything about it (I would not dare to say I do). Now getting a book is not a bad idea, but to get all the information you could possibly want you'd need more of a library than a single book. Most books I have ever seen therefore cover either the basics of Windows programming or a very special topic. And every few month there are some new Software Development Kits coming out which allow you to do even more with Windows. Take the Video for Windows Development Kit (VFWDK) for example. With around 12 MB of hard disk space it is certainly one of the smaller SDKs but it gives you all the tools, include files, libraries and sample application to create, manipulate and play back AVI files(Audio Video Interleaved, the Microsoft format for Video files).

Fortunately there is something that covers everything, and it is even small enough to fit in your room: Microsoft System Development Library. This is a CD full of information about everything you ever wanted to know about Windows including OLE, Visual Basic and Visual Basic for Applications Programming, VBX and OCX Development and all features of Windows 95 and Windows NT. It also contains the complete book "Programming Windows 3.1" by Charles Petzold and all articles ever published in the Microsoft System Journal, a monthly developer magazine, which is also very good source of information. And if you're really interested in what's brewing, you can pop in at Dr. GUI's Espresso Stand which is virtually on the CD (but remember he's not a real doctor!).

To obtain this CD which is updated every three months you've got to subscribe to the Microsoft Developer Network (MSDN). You will then not just get this CD but also all SDK and operating system software you need. To get information about the MSDN write to or e-mail the:

Microsoft Developer Network

One Microsoft Way

Redmond, WA 98052-6399

USA

Fax: (206) 936-7329, Attn: Developer Network

Internet: msdn@microsoft.com

APPENDIX F

The X Window System

ROBERT W. SCHEIFLER
MIT Laboratory for Computer Science
and
JIM GETTYS
Digital Equipment Corporation and MIT Project Athena

An overview of the X Window System is presented, focusing on the system substrate and the low-level facilities provided to build applications and to manage the desktop. The system provides high-performance, high-level, device-independent graphics. A hierarchy of resizable, overlapping windows allows a wide variety of application and user interfaces to be built easily. Network-transparent access to the display provides an important degree of functional separation, without significantly affecting performance, which is crucial to building applications for a distributed environment. To a reasonable extent, desktop management can be custom-tailored to individual environments, without modifying the base system and typically without affecting applications.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.4 [**Operating Systems**]: Communication Management—*network communication*; *terminal management*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; I.3.2 [**Computer Graphics**]: Graphics Systems—*distributed/network graphics*; I.3.4 [**Computer Graphics**]: Graphics Utilities—*graphics packages*; *software support*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*device independence*; *interaction techniques*

General Terms: Design, Experimentation, Human Factors, Standardization

Additional Key Words and Phrases: Virtual terminals, window managers, window systems

1. INTRODUCTION

The X Window System (or simply X) developed at MIT has achieved fairly widespread popularity recently, particularly in the UNIX[1] community. In this paper we present an overview of X, focusing on the system substrate and the low-level facilities provided to build applications and to manage the desktop. In X, this base window system provides high-performance graphics to a hierarchy of resizable windows. Rather than mandate a particular user interface, X provides primitives to support several policies and styles. Unlike most window systems, the base system in X is defined by a *network protocol*: asynchronous

[1]UNIX is a trademark of AT&T Bell Laboratories.

Authors' addresses: R. W. Scheifler, 545 Technology Square, Cambridge, MA 02139; J. Gettys, Project Athena, MIT, Cambridge, MA 02139.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1987 ACM 0730-0301/86/0400-0079 $00.75

stream-based interprocess communication replaces the traditional procedure call or kernel call interface. An application can utilize windows on any display in a network in a device-independent, network-transparent fashion. Interposing a network connection greatly enhances the utility of the window system, without significantly affecting performance. The performance of existing X implementations is comparable to that of contemporary window systems and, in general, is limited by display hardware rather than network communication. For example, 19,500 characters per second and 3500 short vectors per second are possible on Digital Equipment Corporation's VAXStation-II/GPX, both locally and over a local-area network, and these figures are very close to the limits of the display hardware.

X is the result of two separate groups at MIT having a simultaneous need for a window system. In the summer of 1984, the Argus system [16] at the Laboratory for Computer Science needed a debugging environment for multiprocess distributed applications, and a window system seemed the only viable solution. Project Athena [4] was faced with dozens, and eventually thousands, of workstations with bitmap displays and needed a window system to make the displays useful. Both groups were starting with the Digital VS100 display [14] and VAX hardware, but it was clear at the outset that other architectures and displays had to be supported. In particular, IBM workstations with bitmap displays of unknown type were expected eventually within Project Athena. Portability was therefore a goal from the start. Although all of the initial implementation work was for Berkeley UNIX, it was clear that the network protocol should not depend on aspects of the operating system.

The name X derives from the lineage of the system. At Stanford University, Paul Asente and Brian Reid had begun work on the W window system [3] as an alternative to VGTS [13, 22] for the V system [5]. Both VGTS and W allow network-transparent access to the display, using the synchronous V communication mechanism. Both systems provide "text" windows for ASCII terminal emulation. VGTS provides graphics windows driven by fairly high-level object definitions from a structured display file; W provides graphics windows based on a simple display-list mechanism, with limited functionality. We acquired a UNIX-based version of W for the VS100 (with synchronous communication over TCP [24] produced by Asente and Chris Kent at Digital's Western Research Laboratory. From just a few days of experimentation, it was clear that a network-transparent hierarchical window system was desirable, but that restricting the system to any fixed set of application-specific modes was completely inadequate. It was also clear that, although synchronous communication was perhaps acceptable in the V system (owing to very fast networking primitives), it was completely inadequate in most other operating environments. X is our "reaction" to W. The X window hierarchy comes directly from W, although numerous systems have been built with hierarchy in at least some form [11, 15, 18, 28, 30, 32–36]. The asynchronous communication protocol used in X is a significant improvement over the synchronous protocol used in W, but is very similar to that used in Andrew [10, 20]. X differs from all of these systems in the degree to which both graphics functions and "system" functions are pushed back (across the network) as application functions, and in the ability to tailor desktop management transparently.

The next section presents several high-level requirements that we believe a window system must satisfy to be a viable standard in a network environment, and indicates where the design of X fails to meet some of these requirements. In Section 3 we describe the overall X system model and the effect of network-based communication on that model. Section 4 describes the structure of windows, and the primitives for manipulating that structure. Section 5 explains the color model used in X, and Section 6 presents the text and graphics facilities. Section 7 discusses the issues of window exposure and refresh, and their resolution in X. Section 8 deals with input event handling. In Section 9 we describe the mechanisms for desktop management.

This paper describes the version[2] of X that is currently in widespread use. The design of this version is inadequate in several respects. With our experience to date, and encouraged by the number of universities and manufacturers taking a serious interest in X, we have designed a new version that should satisfy a significantly wider community. Section 10 discusses a number of problems with the current X design and gives a general idea of what changes are contemplated.

2. REQUIREMENTS

A window system contains many interfaces. A *programming* interface is a library of routines and types provided in a programming language for interacting with the window system. Both low-level (e.g., line drawing) and high-level (e.g., menus) interfaces are typically provided. An *application* interface is the mechanical interaction with the user and the visual appearance that is specific to the application. A *management* interface is the mechanical interaction with the user, dealing with overall control of the desktop and the input devices. The management interface defines how applications are arranged and rearranged on the screen, and how the user switches between applications; an individual application interface defines how information is presented and manipulated within that application. The *user* interface is the sum total of all application and management interfaces.

Besides applications, we distinguish three major components of a window system. The *window manager*[3] implements the desktop portion of the management interface; it controls the size and placement of application windows, and also may control application window attributes, such as titles and borders. The *input manager* implements the remainder of the management interface; it controls which applications see input from which devices (e.g., keyboard and mouse). The *base window system* is the substrate on which applications, window managers, and input managers are built.

In this paper we are concerned with the base window system of X, with the facilities it provides to build applications and managers. The following requirements for the base window system crystallized during the design of X (a few were not formulated until late in the design process):

1. *The system should be implementable on a variety of displays.* The system should work with nearly any bitmap display and a variety of input devices. Our design focused on workstation-class display technology likely to be available in a

[2] Version 10.

[3] Some people use this term for what we call the base window system; that is not the meaning here.

university environment over the next few years. At one end of the spectrum is a simple frame buffer and monochrome monitor, driven directly by the host CPU with no additional hardware support. At the other end of the spectrum is a multiplane display with color monitor, driven by a high-performance graphics coprocessor. Input devices, such as keyboards, mice, tablets, joysticks, light pens, and touch screens, should be supported.

2. *Applications must be device independent.* There are several aspects to device independence. Most important, it must not be necessary to rewrite, recompile, or even relink an application for each new hardware display. Nearly as important, every graphics function defined by the system should work on virtually every supported display; the alternative, which is to use GKS-style inquire operations [12] to determine the set of implemented functions at run time, leads to tedious case analysis in every application and to inconsistent user interfaces. A third aspect of device independence is that, as far as possible, applications should not need dual control paths to work on both monochrome and color displays.

3. *The system must be network transparent.* An application running on one machine must be able to utilize a display on some other machine, nor should it be necessary for the two machines to have the same architecture or operating system.

There are numerous examples of why this is important: a compute-intensive VLSI design program executing on a mainframe, but displaying results on a workstation; an application distributed over several stand-alone processors, but interacting with a user at a workstation; a professor running a program on one workstation, presenting results simultaneously on all student workstations.

In a network environment, there are certain to be applications that must run on particular machines or architectures. Examples include proprietary software, applications depending on specific architectural properties, and programs manipulating large databases. Such applications still should be accessible to all users. In a truly heterogeneous environment, not all programming languages and programming systems are supported on all machines, and it is very undesirable to have to write an interactive front end in multiple languages in order to make the application generally available. With network-transparent access, this is not necessary; a single front end written in the same language as the application suffices.

One might think that remote display will be extremely infrequent, and that performance is therefore much less important than for local display. Experience at MIT, however, indicates that many users routinely make use of the remote display capabilities in X, and that the performance of remote display is quite important. The desktop display, although physically connected to a single computer, is used as a true *network virtual terminal*; indeed, the idea of an X server (see the next section) built into a Blit-like terminal [23] is an intriguing one.

4. *The system must support multiple applications displaying concurrently.* For example, it should be possible to display a clock with a sweep second hand in one window, while simultaneously editing a file in another window.

5. *The system should be capable of supporting many different application and management interfaces.* No single user interface is "best"; different communities have radically different ideas about user interfaces. Even within a single community, "experts" and "novices" place different demands on an interface. Instead of mandating a particular user interface, the base window system should support a wide range of interfaces.

To achieve this, the system must provide *hooks* (mechanism) rather than *religion* (policy). For example, since menu styles and semantics vary dramatically among different user interfaces, the base window system must provide primitives from which menus can be built, instead of just providing a fixed menu facility.

The system should be designed in such a way that it is possible to implement management policy in a way that is external to the base window system and external to applications. Applications should be largely independent of management policy and mechanism; applications should *react to* management decisions, rather than *direct* those decisions. For example, an application needs to be informed when one of its windows is resized and should react by reformatting the information displayed, but involvement of the application should not be required in order for the user to change the size. Making applications management independent, as well as device independent, facilitates the sharing of applications among diverse cultures.

6. *The system must support overlapping windows, including output to partially obscured windows.* This is in some sense a by-product of the previous requirement, but it is important enough to merit explicit statement. Not all user interfaces allow windows to overlap arbitrarily. However, even interfaces that do not allow application windows to overlap typically provide some form of pop-up menu that overlaps application windows. If such menus are built from windows, then support for overlapping windows must exist.

7. *The system should support a hierarchy of resizable windows, and an application should be able to use many windows at once.* Subwindows provide a clean, powerful mechanism for exporting much of the basic system machinery back to the application for direct use. Many applications make use of their own window-like abstractions; some even implement what is essentially another window system, nested within the "real" window system. It is important to support arbitrary levels of nesting. What is viewed as a single window at one abstraction level may well require multiple subwindows at a lower level. By providing a true window hierarchy, application windows can be implemented as true windows within the system, freeing the application from duplicating machinery such as clipping and input control.

8. *The system should provide high-performance, high-quality support for text, 2-D synthetic graphics, and imaging.* The base window system must provide "immediate" or "transparent" graphics: The application describes the image precisely, and the system does not attempt to second-guess the application. The use of high-level models, whereby the application describes *what* it wants in terms of fairly abstract objects and the system determines *how* best to render the

image, cannot be imposed as the only form of graphics interface. Such models generally fail to provide adequate support for some important class of applications, and different user communities tend to have strong opinions about which model is "best." It is extremely important to provide high-level models, but they should be built in layers on top of the base window system.

Support for 3-D graphics is not listed as a requirement, but this is not to say it is unimportant. We simply have not considered 3-D graphics, owing to lack of expertise and lack of time.

9. *The system should be extensible.* For example, the core system may not support 3-D graphics, but it should be possible to extend the system with such support. The extension mechanism should allow communities to extend the system noncooperatively, yet allow such independent extensions to be merged gracefully.

We believe that a window system must satisfy these requirements to be a viable standard in an environment of high-performance workstations and mainframes connected via high-performance local-area networks. X satisfies most of these requirements, but currently fails to satisfy a few owing to practical considerations of staffing and time constraints: The design and much of the implementation of the base window system were to be handled solely by the first author; it was important to get a working system up fairly quickly; and the immediate applications only required relatively simple text and graphics support. As a result, X is not designed to handle high-end color displays or to deal with input devices other than a keyboard and mouse, some support for high-quality text and graphics is missing, X only provides support for one class of management policy, and no provision has been made for extensions. As discussed in Section 10, these and other problems are being addressed in a redesign of X.

3. SYSTEM MODEL

The X window system is based on a client–server model (see Figure 1); this model follows naturally from requirements 2 and 3 in the previous section. For each physical display, there is a controlling server. A client application and a server communicate over a reliable duplex (8-bit) byte stream. A simple block-stream protocol is layered on top of the byte stream. If the client and server are on the same machine, the stream is typically based on a local interprocess communication (IPC) mechanism; otherwise a network connection is established between the pair. Requiring nothing more than a reliable duplex byte stream (without urgent data) for communication makes X usable in many environments. For example, the X protocol can be used over TCP [24], DECnet [38], and Chaos [19].

Multiple clients can have connections open to a server simultaneously, and a client can have connections open to multiple servers simultaneously. The essential tasks of the server are to multiplex requests from clients to the display, and demultiplex keyboard and mouse input back to the appropriate clients. Typically, the server is implemented as a single sequential process, using round-robin scheduling among the clients, and this centralized control trivially solves many

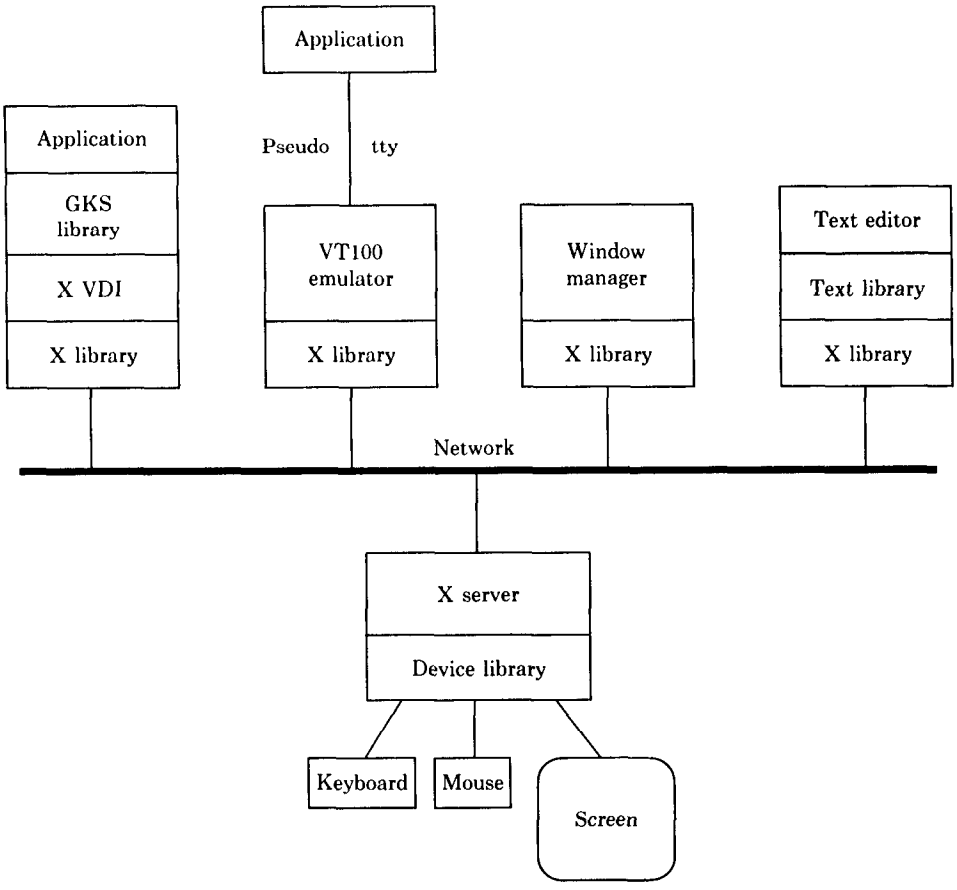


Fig. 1. System structure.

synchronization problems; however, a multiprocess server has also been implemented. Although one might place the server in the kernel of the operating system in an attempt to increase performance, a user-level server process is vastly easier to debug and maintain, and performance under UNIX in fact does not seem to suffer. Similar performance results have been obtained in Andrew [10]. Various tricks are used in both clients and server to optimize performance, principally by minimizing the number of operating system calls [9].

The server encapsulates the base window system. It provides the fundamental resources and mechanisms, and the hooks required to implement various user interfaces. All device dependencies are encapsulated by the server; the communication protocol between clients and server is device independent. By placing all device dependencies on one end of a network connection, applications are truly device independent. The addition of a new display type simply requires the addition of a new server implementation; no application changes are required. Of course, the server itself is designed as device-independent code layered on

top of a device-dependent core, so only the "back end" of the server need be reimplemented for each new display.[4]

3.1 Network Considerations

It is extremely important for the server to be robust with respect to client failures. The server and the network protocol must be designed so that the server never trusts clients to provide correct data. As a corollary, the protocol must be designed in such a way that, if the server ever has to wait for a response from a client, it must be possible to continue servicing other clients. Without this property a buggy client or a network failure could easily cause the entire display to freeze up.

Byte ordering [6] is a standard problem in network communication: When a 16- or 32-bit quantity is transmitted over an 8-bit byte stream, is the most significant byte transmitted first (big-endian byte order) or is the least significant byte transmitted first (little-endian byte order)? Some machines with byte-addressable memory use big-endian order internally, and others use little-endian order. If a single order is chosen for network communication, some machines will suffer the overhead of swapping bytes, even when communicating with a machine using the same internal byte order. Such an approach also means that both parties in the communication must worry about byte order.

The X protocol uses a different approach. The server is designed to accept both big-endian and little-endian connections. For example, using TCP this is accomplished by having the server listen on two distinct ports; little-endian clients connect to the server on one port, and big-endian clients connect on the other. Clients always transmit and receive in their native byte order. The server alone is responsible for byte swapping, and byte swapping only occurs between dissimilar architectures. This eliminates the byte swapping overhead in the most common situations and greatly simplifies the building of client-side interface libraries in various programming languages. X is not unique in its use of this trick; the current VGTS implementation uses the same trick, and similar protocol optimizations have been used in various network-based applications.

Another potential problem in protocol design is word alignment. In particular, some architectures require 16-bit quantities to be aligned on 16-bit boundaries and 32-bit quantities to be aligned on 32-bit boundaries in memory. To allow efficient implementations of the protocol across a spectrum of 16- and 32-bit architectures, the protocol is defined to consist of blocks that are always multiples of 32 bits, and each 16- and 32-bit quantity within a block is aligned on 16- and 32-bit boundaries, respectively.

X is designed to operate in an environment where the interprocess communication round-trip time is between 5 and 50 milliseconds (ms), both for local and for network communication. We also assume that data transmission rates are comparable to display rates; for example, to transmit and display 5000 characters per second, a data rate of approximately 50 kilobits per second (kbits/s) will be needed, and to transmit and display 20,000 characters per second, a data rate of

[4] A back end has been implemented using a programming interface to X itself, such that a complete "recursive" X server executes inside a window of another X server.

approximately 200 kbits/s will be needed. Networks and protocol implementations with these characteristics are now quite commonplace. For example, workstations running Berkeley UNIX, connected via 10-megabit-per-second (Mbit/s) local area networks, typically have round-trip times of 15 to 30 ms, and data rates of 500 kbits/s to 1 Mbit/s.

The round-trip time is important in determining the form of the communication protocol. Text and graphics are the most common requests sent from a client to the server; examples of individual requests might be to draw a string of text or to draw a line. Such requests could be sent either synchronously, in which case the client sends a request only after receiving a reply from the server to the previous request, or they could be sent asynchronously, without the server generating any replies. However, since the requests are sent over a reliable stream, they are guaranteed to arrive and arrive in order, so replies from the server to graphics requests serve no useful purpose. Moreover, with round-trip times over 5 ms, output to the display must be asynchronous, or it will be impossible to drive high-speed displays adequately. For example, at 80 characters per request and a 25-ms round-trip time, only 3200 characters per second can be drawn synchronously, whereas many hardware devices are capable of displaying between 5000 and 30,000 characters per second.

Similarly, polling the server for keyboard and mouse input would be unacceptable in many applications, particularly those written in sequential languages. For example, an application attempting to provide real-time response to input has to poll periodically for input during screen updates. For an application with a single thread of control, this effectively results in synchronous output and consequent performance loss. Hence, input must be generated asynchronously by the server, so that applications need at most perform local polling.

The round-trip time is also important in determining what user interfaces can be supported without embedding them directly in the server. The most important concern is whether remote, application-level mouse tracking is feasible. By *tracking*, we do not mean maintaining the cursor image on the screen as the user moves the mouse; that function is performed autonomously by the X server, often directly in hardware. Rather, applications track the mouse by animating some other image on the screen in real time as the mouse moves. For round-trip times under 50 ms, tracking is perfectly reasonable, driven either by motion events generated by the server or by continuous polling from the application. With a refresh occurring up to 30 times every second, remote tracking is demonstrably "instantaneous" with mouse motion.

For tracking to be effective, however, relatively little time can be spent updating the display at each movement, so typically only relatively small changes can be made to the screen while tracking. This is certainly the case for simple tracking, such as rubber banding window outlines and highlighting menu items. It might be argued that the ability to run application-specific code in the server is required for acceptable hand–eye coordination during more complex tracking. For example, NeWS [31] provides such a mechanism in a novel way. However, we are not convinced there are sufficient benefits to justify such complexity. Typically, complex tracking is bound intimately to application-specific data structures and knowledge representations. The information needed by the front end for tracking

is intertwined with the information needed by the back end for the "real" work; the information cannot be reasonably separated or duplicated. It is simply unreasonable to believe that applications will download large, complex front ends into a server; communication round-trip times are a reality that cannot be escaped.

3.2 Resources

The basic resources provided by the server are windows, fonts, mouse cursors, and offscreen images; later sections describe each of these. Clients request creation of a resource by supplying appropriate parameters (such as the name of the font); the server allocates the resource and returns a 29-bit[5] unique identifier used to represent it. The use and interpretation of a resource identifier are independent of any network connection. Any client that knows (or guesses) the identifier for a resource can use and manipulate the resource freely, even if it was created by another client. This capability is required to allow window managers to be written independently of applications, and to allow multiprocess applications to manipulate shared resources. However, to avoid problems associated with clients that fail to clean up their resources at termination (which is all too common in operating systems where users can unilaterally abort processes), the maximum lifetime of a resource is always tied to the connection over which it was created. Thus, when a client terminates, all of the resources it created are destroyed automatically.

Access control is performed only when a client attempts to establish a connection to the server; once the connection is established, the client can freely manipulate any resource. Since accidental manipulation of some other client's resource is extremely unlikely (both in theory and in practice), we believe introducing access control on a per-resource basis would only serve to decrease performance, not to significantly increase security or robustness. The current access control mechanism is based simply on host network addresses, as this information is provided by most network stream protocols, and there seems to be no widely used or even widely available user-level authentication mechanism. Host-based access control has proved to be marginally acceptable in a workstation environment, but is rather unacceptable for time-shared machines.[6]

Each client-generated protocol request is a simple data block consisting of an opcode, some number of fixed-length parameters, and possibly a variable-length parameter. For example, to display text in a window, the fixed-length parameters include the drawing color and the identifiers for the window and the font, and the variable-length parameter is the string of characters. All operations on a resource explicitly contain the identifier of the resource as a parameter. In this way, an application can multiplex use of many windows over a single network connection. This multiplexing makes it easy for the client to control the time order of updates to multiple windows; if a separate stream was used for each window, time order could not be controlled without strong guarantees from the stream mechanism. Similarly, each input event generated by the server contains

[5] To simplify implementation in languages built with garbage collection, high-order bits are not used.
[6] It is interesting that *professors* at MIT have argued vociferously to disable all access control.

the identifier of the window in which the event occurred. Multiplexing over a single stream allows the client to act on events from multiple windows in correct time order; again, the use of a stream per window would not allow such ordering, even if the events carry timestamps.

Numerous UNIX-based window systems [17, 20, 21, 30, 36] use file or channel descriptors to represent windows; window creation involves an interaction with the operating system, which results in the creation of such a descriptor. Typically, this means the window cannot be named (and hence cannot be shared) by programs running on different machines, and perhaps not even by programs running on the same machine. More serious, there is often a severe restriction on the number of active descriptors a process may have: 20 on older systems and usually 64 on newer systems. The use of 50 or more windows (albeit nested inside a single top-level window) is quite common in X applications. The use of a single connection, over which an arbitrary number of windows can be multiplexed, is clearly a better approach.

4. WINDOW HIERARCHY

The server supports an arbitrarily branching hierarchy of rectangular windows. At the top is the *root* window, which covers the entire screen. The *top-level* windows of applications are created as subwindows of the root window. The window hierarchy models the now-familiar "stacks of papers" desktop. For a given window, its subwindows can be stacked in any order, with arbitrary overlaps. When window W1 partially or completely covers window W2, we say that W1 *obscures* W2. This relationship is not restricted to siblings; if W1 obscures W2, then W1 may also obscure subwindows of W2. A window also obscures its parent. Window hierarchies never interleave; if window W1 obscures sibling window W2, then subwindows of W2 never obscure W1 or subwindows of W1. A window is not restricted in size or placement by the boundaries of its parent, but a window is always visibly clipped by its parent: Portions of the window that extend outside the boundaries of the parent are never displayed and do not obscure other windows. Finally, a window can be either *mapped* or *unmapped*. An unmapped window is never visible on the screen; a mapped window can only be visible if all of its ancestors are also mapped.

Output to a leaf window (one with no subwindows) is always clipped to the visible portions of the window; drawing on such a window never draws into obscuring windows. Output to a window that contains subwindows can be performed in two modes. In *clipped* mode the output is clipped normally by all obscuring windows (including subwindows), but in *draw-through* mode the output is not clipped by subwindows. For example, draw-through mode is used on the root window during window management, tracking the mouse with the outline of a window to indicate how the window is to be moved or resized. If clipped mode were used instead, the entire outline would not be visible.

The coordinate system is defined with the X axis horizontal and the Y axis vertical. Each window has its own coordinate system, with the origin at the upper left corner of the window. Having per-window coordinate systems is crucial, particularly for top-level windows; applications are almost always designed to be insensitive to their position on the screen, and having to worry about race

conditions when moving windows would be a disaster. The coordinate system is discrete: Each pixel in the window corresponds to a single unit in the coordinate system, with coordinates centered on the pixels, and all coordinates are expressed as integers in the protocol. We believe fractional coordinates are not required at the protocol level for the raster graphics provided in X (see Section 6), although they may be required for high-end color graphics, such as antialiasing. The aspect ratio of the screen is not masked by the protocol, since we believe that most displays have a one-to-one aspect ratio; in this regard X is arguably device dependent.

Although the coordinate system is discrete at the protocol level, continuous or alternate-origin coordinate systems certainly can be used at the application level, but client-side libraries must eventually translate to the discrete coordinates defined by the protocol. In this way, we can ignore the many variations in floating-point (or even fixed-point) formats among architectures. Further, the coordinates can be expressed in the protocol as 16-bit quantities, which can be manipulated efficiently in virtually every machine/display architecture and which minimizes the number of data bytes transmitted over the network. The use of 16-bit quantities does have a drawback, in that some applications (particularly CAD tools) like to perform zoom operations simply by scaling coordinates and redrawing, relying on the window system to clip appropriately. Since scaling quickly overflows 16 bits, additional clipping must be performed explicitly by such applications.

A window can optionally have a *border*, a shaded outer frame maintained explicitly by the X server. The origin of the window's coordinate system is inside the border, and output to the window is clipped automatically so as not to extend into the border. The presence of borders slightly complicates the semantics of the window system; for simplicity we ignore them in the remainder of this paper.

The basic operations on window structure are straightforward. An unmapped window is created by specifying the parent window, the position within the parent of the upper left corner of the new window, and the width and height (in coordinate units) of the new window. A window can be destroyed, in which case all windows below it in the hierarchy are also destroyed. A window can be mapped and unmapped, without changing its position. A window can be moved and resized, including being moved and resized simultaneously. A window can also be "depthwise" raised to the top or lowered to the bottom of the stack with respect to its siblings, without changing its coordinate position. Currently mapping or configuring a window forces the window to be raised. This restriction appeared to simplify the server implementation but also happened to match the basic management interface we expected to build. This restriction will be eliminated in the next version.

The windows described above are the usual *opaque* windows. X also provides *transparent* windows. A transparent window is always invisible on the screen and does not obscure output to, or visibility of, other windows. Output to a transparent window is clipped to that window but is actually drawn on the parent window. Thus, for output, a transparent window is simply a clipping rectangle that can be applied to restrict output within a (parent) window. Input processing for transparent and opaque windows is identical, as described in Section 8. In

Section 10 we argue that most uses of transparent windows are better satisfied with other mechanisms. Therefore, for simplicity, we ignore transparent windows in the rest of this paper.

The X server is designed explicitly to make windows inexpensive. Our goal was to make it reasonable to use windows for such things as individual menu items, buttons, and even individual items in forms and spreadsheets. As such, the server must deal efficiently with hundreds (though not necessarily thousands) of windows on the screen simultaneously. Experience with X has shown that many implementors find this capability extremely useful, particularly when building extensible tool kits.

5. COLOR

The screen is viewed as two dimensional, with an N-bit *pixel* value stored at each coordinate. The number of bits in a pixel value and how a value translates into a color depend on the hardware. X is designed to support two types of hardware: monochrome and pseudocolor. A monochrome display has 1 bit per pixel, and the two values translate into black and white. Pseudocolor displays typically have between 4 and 12 bits per pixel; the pixel value is used as an index into a color map, yielding red, green, and blue intensities. The color map can be changed dynamically, so that a given pixel value can represent different colors over time. Gray scale is viewed as a degenerate case of pseudocolor.

We desire a design matching most display hardware, while abstracting differences in such a way that programmers do not have to double- or triple-code their applications to cover the spectrum. We also want multiple applications to coexist within a single color map, so that applications always show true color on the screen. To allow this and to keep applications device independent, pixel values should not be coded explicitly into applications. Instead, the server must be responsible for managing the color map, and color map allocation must be expressed in hardware-independent terms.

All graphics operations in X are expressed in terms of pixel values. For example, to draw a line, one specifies not only the coordinates of the endpoints, but the pixel value with which to draw the line. (Logic functions and plane-select masks are also specified, as described in Section 6.) On a monochrome display, the only two pixel values are 0 and 1, which are (somewhat arbitrarily) defined to be black and white, respectively. On a pseudocolor display, pixel values 0 and 1 are preallocated by the server for use as "black" and "white" so that monochrome applications display correctly on color displays. Of course, the actual colors need not be black and white, but can be set by the user.

There are two ways for a client to obtain pixel values. In the simplest request the client specifies red, green, and blue color values, and the server allocates an arbitrary pixel value and sets the color map so that the pixel value represents the closest color the hardware can provide. The color map entry for this pixel value cannot be changed by the client, so, if some other client requests an equivalent color, the server is free to respond with the same pixel value. Such sharing is important in maximizing use of the color map. To isolate applications from variations in color representation among displays (e.g., due to the standard of illumination used for calibration), the server provides a color database that

clients can use to translate string names of colors into red, green, and blue values tailored for the particular display.

The second request allocates writable map entries. This mechanism was designed explicitly for X; we are not aware of a comparable mechanism in any other window system. The client specifies two numbers, C and P, with C positive and P nonnegative; the request can be expressed as "allocate C colors and P planes." The total number of pixel values allocated by the server is $C \times 2^P$. The values passed back to the client consist of C base pixel values and a plane mask containing P bits. None of the base pixel values have any 1 bits in common with the plane mask, and the complete set of allocated pixel values is obtained by combining all possible combinations of 1 bits from the plane mask with each of the base pixel values. The client can optionally require the P planes to be contiguous, in which case all P bits in the plane mask will be contiguous.

There are three common uses of this second request. One is simply to allocate a number of "unrelated" pixel values; in this case P will be 0. A second use is in imaging applications, where it is convenient to be able to perform simple arithmetic on pixel values. In this case a contiguous block of pixel values is allocated by setting C to 1 and P to the log (base 2) of the number of pixel values required, and requesting contiguous allocation. Arithmetic on the pixel values then requires at most some additional shift and mask operations.

A third form of allocation arises in applications that want some form of overlay graphics, such as highlighting or outlining regions. Here the requirement is to be able to draw and then erase graphics without disturbing existing window contents. For example, suppose an application typically uses four colors, but needs to be able to overlay a rectangle outline in a fifth color. An allocation request with C set to 4 and P set to 1 results in two groups of four pixel values. The four base pixel values are assigned the four normal colors, and the four alternate pixel values are all assigned the fifth color. Overlay graphics can then be drawn by restricting output (see the next section) to the single bit plane specified in the mask returned by the color allocation. Turning bits in this plane on (to 1's) changes the image to the fifth color, and turning them off reverts the image to its original color.

6. GRAPHICS AND TEXT

Graphics operations are often the most complex part of any window system, simply because so many different effects and variations are required to satisfy a wide range of applications. In this section we sketch the operations provided in X so that the basic level of graphics support can be understood. The operations are essentially a subset of the Digital Workstation Graphics Architecture; the VS100 display [14] implements this architecture for 1-bit pixel values. The set of operations was purposely kept simple in order to maximize portability.

Graphics operations in X are expressed in terms of relatively high-level concepts, such as lines, rectangles, curves, and fonts. This is in contrast to systems in which the basic primitives are to read and write individual pixels. Basing applications on pixel-level primitives works well when display memory can be mapped into the application's address space for direct manipulation. However, both display hardware and operating systems exist for which such

direct access is not possible, and emulating pixel-level manipulations in such an environment results in extremely poor performance. Expressing operations at a higher level avoids such device dependencies, as well as potential problems with network bandwidth. With high-level operations, a protocol request transmitted as a small number of bits over the network typically affects 10–100 times as many pixels on the screen.

6.1 Images

Two forms of offscreen images are supported in X: bitmaps and pixmaps. A bitmap is a single plane (bit) rectangle. A pixmap is an N-plane (pixel) rectangle, where N is the number of bits per pixel used by the particular display. A bitmap or pixmap can be created by transmitting all of the bits to the server; a pixmap can also be created by copying a rectangular region of a window. Bitmaps and pixmaps of arbitrary size can be created. Transmitting very large (or deep) images over a network connection can be quite slow; however, the ability to make use of shared memory in conjunction with the IPC mechanism would help enormously when the client and server are on the same machine.

The primary use of bitmaps is as masks (clipping regions). Several graphics requests allow a bitmap to be used as a clipping region, as in [37]. Bitmaps are also used to construct cursors, as described in Section 8. Pixmaps are used for storing frequently drawn images and as temporary backing-store for pop-up menus (as described in Section 8). However, the principal use of pixmaps is as tiles, that is, as patterns that are replicated in two dimensions to cover a region. Since there are often hardware restrictions as to what tile shapes can be replicated efficiently, guaranteed shapes are not defined by the X protocol. An application can query the server to determine what shapes are supported, although to date most applications simply assume 16-by-16 tiles are supported. A better semantics is to support arbitrary shapes but allow applications to query which shapes are most efficient.

The tiling origin used in X is almost always the origin of the destination window. That is, if enough tiles have been laid out, one tile would have its upper left corner at the upper left corner of the window. In this way, the contents of the window are independent of the window's position on the screen, and the window can be moved transparently to the application.

Servers vary widely in the amount of offscreen memory provided. For example, some servers limit offscreen memory to that accessible directly to the graphics processor (typically one to three times the size of screen memory), and fonts and other resources are allocated from this same pool. Other servers utilize their entire virtual address space for offscreen memory. Since offscreen memory for images is finite, an explicit part of the X protocol is the possibility that bitmap or pixmap creation can fail. Depending on the intended use of the image, the application may or may not be able to cope with the failure. For example, if the image is being stored simply to speed up redisplay, the application can always transmit the image directly each time (see below). If the image is to be a temporary backing-store for a window, the application can fall back on normal exposure processing (as described in Section 7). Servers should be constructed in such a way as to virtually guarantee sufficient memory (e.g., by caching images) for

creating at least small tiles and cursors, although this is not true in current implementations.

6.2 Graphics

All graphics and text requests include a logic function and a plane-select mask (an integer with the same number of bits as a pixel value) to modify the operation. All 16 logic functions are provided, although in practice only a few are ever used. Given a source and destination pixel, the function is computed bitwise on corresponding bits of the pixels, but only on bits specified in the plane-select mask. Thus the result pixel is computed as

((source FUNC destination), AND mask) OR (destination AND (NOT mask)).

The most common operation is simply replacing the destination with the source in all planes.

The simplest graphics request takes a single source pixel value and combines it with every pixel in a rectangular region of a window. Typically, this is used to fill a region with a color, but by varying the logic function or masks, other effects can be achieved. A second request takes a tile, effectively constructs a tiled rectangular source with it, and then combines the source with a rectangular region of a window.

An arbitrary image can be displayed directly, without first being stored off-screen. For monochrome images, the full contents of a bitmap are transmitted, along with a pair of pixel values; the image is displayed in a region of a window with those two colors. For color images, the full contents of a pixmap can be transmitted and displayed. In order to avoid requiring inordinate buffer space in the server, very large images must be broken into sections on the client side and displayed in separate requests.

The CopyArea request allows one region of a window to be moved to (or combined with) another region of the same window. This is the usual *bitblt*, or "bit block transfer" operation. The source and destination are given as rectangular regions of the window; the two regions have the same dimensions. The operation is such that overlap of the source and destination does not affect the result.

X provides a complex primitive for line drawing. It provides for arbitrary combinations of straight and curved segments, defining both open and closed shapes. Lines can be *solid*, by drawing with a single source pixel value, *dashed*, by alternately drawing with a single source pixel value and not drawing, and *patterned*, by alternately drawing with two source pixel values. Lines are drawn with a rectangular brush. Clients can query the server to determine what brush shapes are supported; a better semantics would be to support arbitrary shapes but allow applications to query which shapes are most efficient.

A final request allows an arbitrary closed shape (such as could be specified in the line-drawing request) to be filled with either a single source pixel value or a tile. For self-intersecting shapes, the even–odd rule is used: A point is inside the shape if an infinite ray with the point as origin crosses the path an odd number of times.

6.3 Text

For high-performance text, X provides direct support for bitmap fonts. A font consists of up to 256 bitmaps; each bitmap in a font has the same height but can vary in width. To allow server-specific font representations, clients "create" fonts by specifying a name rather than by downloading bitmap images into the server. An application can use an arbitrary number of fonts, but (as with all resources) font allocation can fail for lack of memory. A reasonably implemented server should support an essentially unbounded number of fonts (e.g., by caching), but some existing server implementations are deficient in this respect. Unlike Andrew [10], no heuristics are applied by the server when resolving a name to a font; specific communities or applications may demand a variety of heuristics, and as such they belong outside the base window system. Also unlike Andrew, the X server is not free to dynamically substitute one font for another; we do not believe such behavior is necessary or appropriate in the base window system.

A string of text can be displayed by using a font either as a mask or as a source. When a font is used as a mask, the foreground (the 1 bits in the bitmap) of each character is drawn with a single source pixel value. When a font is used as a source, the entire image of each character is drawn, using a pair of pixel values. Source font output is provided specifically for applications using fixed-width fonts in emulating traditional terminals.

To support "cut-and-paste" operations between applications, the server provides a number of buffers into which a client can read and write an arbitrary string of bytes. (This mechanism was adopted from Andrew.) Although these buffers are used principally for text strings, the server imposes no interpretation on the data, so cooperating applications can use the buffers to exchange such things as resource identifiers and images.

7. EXPOSURES

Given that output to obscured windows is possible, the issue of *exposure* must be addressed. When all (or a piece) of an obscured window again becomes visible (e.g., as the result of the window being raised), is the client or the server responsible for restoring the contents of the window? In X, it is the responsibility of the client. When a region of a window becomes exposed, the server sends an asynchronous event to the client specifying the window and the region that have been exposed; the rest is up to the application. A trivial application might simply redraw the entire window; a more sophisticated application would only redraw the exposed region.

Why is the client responsible? Because X imposes no structure on or relationships between graphics operations from a client, there are only two basic mechanisms by which the server might restore window contents: by maintaining display lists and by maintaining offscreen images. In the first approach, the server essentially retains a list of all output requests performed on the window. When a region of the window becomes exposed, the server reexecutes either all requests to the entire window or only requests that affect the region while clipping the output to that region. In the alternative approach, when a window

becomes obscured, the server saves the obscured region (or perhaps the entire window) in offscreen memory. All subsequent output requests are executed not only to the visible regions of the window, but to the offscreen image as well. When an obscured region becomes visible again, the offscreen copy is simply restored.

We believe that neither server-based approach is acceptable. With display lists, the server is unlikely to have any reasonable notion of when later output requests nullify earlier ones. Either the display list becomes unmanageably long, and a refresh that should appear nearly instantaneous instead appears as an extended replay, or the server spends a significant length of time pruning the display list, and normal-case performance is considerably reduced. One problem with the offscreen image approach is (virtual) memory consumption: On a 1024-by-1024 eight-plane display, just one full-screen image requires 1 megabyte (Mbyte) of storage, and multiple overlapping windows could easily require many times that amount. Another problem is that the cost of the implementation can be prohibitive. Consider, for example, the QDSS display [7], which has a graphics coprocessor. In the QDSS, display memory is inaccessible to the host processor. In addition, the coprocessor cannot perform operations in host memory and has relatively little offscreen memory of its own. The only viable way to maintain offscreen images for displays like the QDSS may be to emulate the coprocessor in software. It can easily take tens of thousands of lines of code to emulate a coprocessor, and such emulation may execute orders of magnitude slower than the coprocessor.

Our belief is that many applications can take advantage of their own information structures to facilitate rapid redisplay, without the expense of maintaining a distinct display structure or backing-store in the client or the server, and often with even better performance. (Sapphire [21] permits client refresh for this reason.) For example, a text editor can redisplay directly from the source, and a VLSI editor can redisplay directly from the layout and component definitions. Many applications will be built on top of high-level graphics libraries that automatically maintain the data structures necessary to implement rapid redisplay. For example, the structured display file mechanism in VGTS could be supported in a client library. Of course, pushing the responsibility back on the application may not simplify matters, particularly when retrofitting old systems to a new environment. For example, the current GKS design does not quite provide adequate hooks for automatic, system-generated refresh of application windows, nor does it provide an adequate mechanism for forcing refresh back on the application.

Relying on client-controlled refresh also derives from window management philosophy. Our belief is that applications cannot be written with fixed top-level window sizes built in. Rather, they must function correctly with almost any size and continue to function correctly as windows are dynamically resized. This is necessary if applications are to be usable on a variety of displays under a variety of window management policies. (Of course, an application may need a minimum size to function reasonably and may prefer the width or height to be a multiple of some number; X allows the client to attach a resize hint to each window to inform window managers of this.) Our belief is that most applications, for one

reason or another, will already have code for performing a complete redisplay of the window, and that it is usually straightforward to modify this code to deal with partial exposures. Similar arguments were used in the design of both Andrew and Mex and confirmed by experience [10, 25, 26].

This is not to argue that the server should never maintain window contents, only that it should not be *required* to maintain contents. For complex imaging and graphics applications, efficient maintenance by the server may be critical for acceptable performance of window management functions. There is nothing inherent in the X protocol that precludes the server from maintaining window contents and not generating exposure events. In the next version of X, windows will have several attributes to advise the server as to when and how contents should be maintained.

In X, clients are never informed of what regions are obscured, only of what regions have become visible. Thus, clients have insufficient information for optimizing output by only drawing to visible regions. However, we feel this is justified on two grounds. First, realistically, users seldom stack windows such that the active ones are obscured, so there is little point in complicating applications to optimize this case. More important, allowing applications to restrict output to only visible regions would conflict with the desire to have the server maintain obscured regions automatically when possible.

An interesting complication with the CopyArea request (described in Section 6) arises when client refresh is decided on. If part of the source region of the CopyArea is obscured, then not all of the destination region can be updated properly, and the client must be notified (with an exposure event) so that it can correct the problem. Since output requests are asynchronous, care must be taken by the application to handle exposure events when using CopyArea. In particular, if a region is exposed and an event sent by the server, a subsequent CopyArea may move all or part of the region before the event is actually received by the application. Several simple algorithms have been designed to deal with this situation, but we do not present them here.

Client refresh raises a visual problem in a network environment. When a region of a window becomes exposed, what contents should the server initially place in the window? In a local, tightly coupled environment, it might be perfectly reasonable to leave the contents unaltered, because the client can almost instantaneously begin to refresh the region. In a network environment, however (and even in a local system where processes can get "swapped out" and take considerable time to swap back in), inevitable delays can lead to visually confusing results. For example, the user may move a window and see two images of the window on the screen for a significant length of time, or resize a window and see no immediate change in the appearance of the screen.

To avoid such anomalies in X, clients must define a *background* for every window. The background can be a single color, or it can be a tiling pattern. Whenever a region of a window is exposed, the server immediately paints the region with the background. Users therefore see window shapes immediately, even if the "contents" are slow to arrive. Of course, many application windows have some notion of a background anyway, so having the server initialize with a background seldom results in extraneous redisplay. In fact, many nonleaf

windows typically contain nothing but a background, and having the server paint that background frees the applications from performing any redisplay at all to those windows.

Although we believe client-generated refresh is acceptable most of the time, it does not always perform well with momentary pop-up menus, where speed is at a premium. To avoid potentially expensive refresh when a menu is removed from the screen, a client can explicitly copy the region to be covered by the menu into offscreen memory (within the server) before mapping the menu window. A special unmap request is used to remove the menu: It unmaps the window without affecting the contents of the screen or generating exposure events. The original contents are then copied back onto the screen. In addition, the client usually *grabs* the server for the entire sequence, using a request that freezes all other clients until a corresponding ungrab request is issued (or the grabbing client terminates). Without this, concurrent output from other clients to regions obscured by the menu would be lost. Although freezing other clients is, in general, a poor idea, it seems acceptable for momentary menus.

8. INPUT

We now turn to a discussion of input events, but first we briefly describe the support for mouse cursors. Clients can define arbitrary shapes for use as mouse cursors. A cursor is defined by a source bitmap, a pair of pixel values with which to display the bitmap, a mask bitmap that defines the precise shape of the image, and a coordinate within the source bitmap that defines the "center" or "hot spot" of the cursor. Cursors of arbitrary size can be constructed, although only a portion of the cursor may be displayed on some hardware. Clients can query the server to determine what cursor sizes are supported, but existing applications typically just assume a 16-by-16 image can always be displayed. Cursors also can be constructed from character images in fonts; this provides a simple form of named indirection, allowing custom-tailoring to each display without having to modify the applications.

A window is said to *contain* the mouse if the hot spot of the cursor is within a visible portion of the window or one of its subwindows. The mouse is said to be *in* a window if the window, but no subwindow, contains the mouse. Every window can have a mouse cursor defined for it. The server automatically displays the cursor of whatever window the mouse is currently in; if the window has no cursor defined, the server displays the cursor of the closest ancestor with a cursor defined.

Input is associated with windows. Input to a given window is controlled by a single client, which need not be the client that created the window. Events are classified into various types, and the controlling client selects which types are of interest to it. Only events matching in type with this selection are sent to the client. When an input event is generated for a window and the controlling client has not selected that type, the server *propagates* the event to the closest ancestor window for which some client has selected the type, and sends the event to that client instead. Every event includes the window that had the event type selected; this window is called the *event window*. If the event has been propagated, the

event also includes the next window down in the hierarchy between the event window and the original window on which the event was generated.

8.1 The Keyboard

For the keyboard, a client can selectively receive events on the press or release of a key. Keyboard events are not reported in terms of ASCII character codes; instead, each key is assigned a unique code, and client software must translate these codes into the appropriate characters. The mapping from keycaps to keycodes is intended to be "universal" and predefined; a given keycap has the same keycode on all keyboards. Applications generally have been written to read a "keymap file" from the user's home directory so that users can remap the keyboard as they see fit.

The use of coded keys is secondary to the ability to detect both up and down transitions on the keyboard. For example, a common trick in window systems is for mouse button operations to be affected by keyboard *modifiers* such as the Shift, Control, and Meta keys. A useful feature of the Genera [34] system is the use of a "mouse documentation line," which changes dynamically as modifiers are pressed and released, indicating the function of the mouse buttons. A base window system must provide this capability. Transitions are not only useful on modifiers; various applications for systems other than X have been designed to use "chords" (groups of keys pressed simultaneously), and again the window system should support them.

The keyboard is always *attached* to some window (typically the root window or a top-level window); we call this window the *focus* window. A request can be used (usually by the input manager) to attach the keyboard to any window. The window that receives keyboard input depends on both the mouse position and the focus window. If the mouse is in some descendant of the focus window, that descendant receives the input. If the mouse is not in a descendant of the focus window, then the focus window receives the input, even if the mouse is outside the focus window. For applications that wish to have the mouse state modify the effect of keyboard input, a keyboard event contains the mouse coordinates, both relative to the event window and global to the screen, as well as the state of the mouse buttons.

To provide a reasonable user interface, keyboard events also contain the state of the most common modifier keys: Shift, ShiftLock, Control, and Meta. Without this information, anomalous behavior can result. If the user switches windows while modifier keys are down, the new client must somehow determine which modifiers are down. Placing the modifier state in the keyboard events solves such problems and also has another benefit: Most clients do not have to maintain their own shadow of the modifier state and so often can completely ignore key release events. However, there is a conflict between this server-maintained state and client-maintained keyboard mappings. In particular, clients cannot use nonstandard keys as modifiers or chords without the possibility of anomalies, such as those described above. We believe the correct solution (not yet supported in X) is for the server to maintain a bit mask reflecting the full state of the keyboard and allow clients to read this mask. An application using chords or

nonstandard modifiers would request the server to send this mask automatically whenever the mouse has entered the application's window.

8.2 The Mouse

The X protocol is (somewhat arbitrarily) designed for mice with up to three buttons. An application can selectively receive events on the press or release of each button. Each event contains the current mouse coordinates (both local to the window and global to the screen), the current state of all buttons and modifier keys, and a timestamp that can be used, for example, to decide when a succession of clicks constitutes a double or triple click. An application can also choose to receive mouse motion events, either whenever the mouse is in the window or only when particular buttons have also been pressed. The application cannot control the granularity of the reporting, nor is any minimum granularity guaranteed. In fact, typical server implementations make an effort to compact motion events in order to minimize system overhead and wired memory in device drivers. Thus X may not serve adequately for fine-grained tracking, such as in fast moving freehand drawing applications.

Even with motion compaction, servers can generate considerable numbers of motion events. If an application attempts to respond in real time to every event, it can easily get far behind relative to the actual position of the mouse. Instead, many applications simply treat motion events as hints. When a motion event is received, the event is simply discarded, and the client then explicitly queries the server for the current mouse position. While waiting for the reply, more motion events may be received; these are also discarded. The client then reacts on the basis of the queried mouse position. The advantage of this scheme over continuously polling the mouse position is that no CPU time is consumed while the mouse is stationary.

Clients can also receive an event each time the mouse enters or leaves a window. This can be particularly useful in implementing menus. For example, each menu item can be placed in a separate subwindow of the overall menu window. When the mouse enters a subwindow, the item is highlighted in some fashion (e.g., by inverting the video sense), and when the mouse leaves the window, the item is restored to normal. Implementing a menu in this manner requires considerably less CPU overhead than continuously polling the mouse, and also less overhead than using motion events, since most motion events would be within windows and thus uninteresting.

Owing to the nature of overlapping windows and because continuous tracking by the server is not guaranteed, the mouse may appear to move instantaneously between any pair of windows on the screen. Certainly, the window the mouse was in should be notified of the mouse leaving, and the window the mouse is now in should be notified of the mouse entering. However, all of the "in between" windows in the hierarchy may also be interested in the transition. This is useful in simplifying the structure of some applications and is necessary in implementing certain kinds of window managers and input managers. Thus, when the mouse moves from window A to window B, with window W as their closest (least) common ancestor, all ancestors of A below W also receive leave events, and all ancestors of B below W receive enter events.

It might be argued that, except for mouse motion events, events are infrequent enough for the server to always send all events to the client and eliminate the complexity of selecting events. However, some applications are written with interrupt-driven input; events are received asynchronously and cause the current computation to be suspended so that the input can be processed. For example, a text editor might use interrupt-driven input, with the normal computation being redisplay of the window. The receipt of extraneous input events (e.g., key release events) can cause noticeable "hiccups" in such redisplay.

9. INPUT AND WINDOW MANAGEMENT

There are two basic modes of keyboard management: *real-estate* and *listener*. In real-estate mode, the keyboard "follows" the mouse; keyboard input is directed to whatever window the mouse is in. In listener mode, keyboard input is directed to a specific window, independent of the mouse position. A few systems provide only real-estate mode [2], some only listener mode [11, 18, 21, 25, 33, 34], and a few provide both [10, 30], although the mode may not be changeable during a session. Both modes are supported in X, and the mode can be changed dynamically. Real-estate mode is the default behavior, with the root window as the focus window, as described in the previous section. An input manager can also make some other (typically top-level) window the focus window, yielding listener mode. Note, however, that, in listener mode in X, the client controlling the focus window can still get real-estate behavior for subwindows, if desired; this capability has proved useful in several applications.

The primary function of a window manager is reconfiguration: restacking, resizing, and repositioning top-level windows. The configuration of nested windows is assumed to be application specific, and under control of the applications. There are two broad categories of window managers: *manual* and *automatic*. A manual window manager is "passive" and simply provides an interface to allow the user to manipulate the desktop; windows can be resized and reorganized at will. The initial size and position of a window typically (but not always) are under user or application control. Automatic window managers are "active" and operate for the most part without human interaction; size and position at window creation and reconfiguration at window destruction are chosen by the system. Automatic managers typically tile the screen with windows such that no two windows overlap, automatically adjusting the layout as windows are created and destroyed. Several systems [10, 18, 27, 36] provide automatic management plus limited manual reconfiguration capability.

Existing window managers for X are manual. Automatic management that is transparent to applications cannot be accomplished reasonably in X; future support for automatic management is discussed in Section 10. In the current X design, clients are responsible for initially sizing and placing their top-level windows, not window managers. In this way, applications continue to work when no window manager is present. Typically, the user either specifies geometry information in the application command line or uses the mouse to sweep out a rectangle on the screen. (For the latter, the application grabs the mouse, as described below.)

9.1 Mouse-Driven Management

Existing managers are primarily mouse driven and are based on the ability to "steal" events. Specifically, a manager (or any other client) can *grab* a mouse button in combination with a set of modifier keys, with the following effect: Whenever the modifier keys are down and the button is pressed, the event is reported to the grabbing client, regardless of what window the mouse is in. All mouse-related events continue to be sent to that client until the button is released. As part of the grab, the client also specifies a mouse cursor to be used for the duration of the grab and a window to be used as the event window. A manager specifies the root window as the event window when grabbing buttons; with the event propagation semantics described in Section 8, the grabbed events contain not only the global mouse coordinates, but also the top-level application window (if any) containing the mouse. This is sufficient information to manipulate top-level windows.

This button-grab mechanism has enabled several different management interfaces to be built, including a "programmable" interface [8] that allows the user to assign individual commands or user-defined menus of commands to any number of button/modifier combinations. For example, a button click (press and release without intervening motion) might be interpreted as a command to raise or lower a window, or to attach the keyboard; a press/motion/release sequence might be interpreted as a command to move a window to a new position; or a button press might cause a menu to pop up, with the selection indicated by the mouse position at the release of the button. By allowing both specific commands and menus to be bound to buttons, a range of interfaces can be constructed to satisfy both "expert" and "novice" users.

Another form of manager simply displays a static menu bar along the top of the screen, with items for such operations as moving a window and attaching the keyboard. The menu is used in combination with a mouse-grab primitive, with which a client can unilaterally grab the mouse and then later explicitly release it; during such a mouse grab, events are redirected to the grabbing client, just as for button grabs. When the user clicks on a menu bar item with any button, the manager unilaterally grabs the mouse. The user then uses the mouse to execute the specific command. For example, having clicked on the "move" item, the user indicates the window to be moved by placing the mouse in the window and pressing a button and then indicates the new position by moving the mouse and releasing the button. The manager then releases the mouse.

9.2 Icons

One important "resizing" operation performed by a window manager is transforming a window into a small icon and back again. In X, icons are merely windows. Transforming a window into an icon simply involves unmapping the window and mapping its associated icon. The association between a window and its icon is maintained in the server, rather than in the window manager, and either the application or the manager can provide the icon. In this way, the manager can provide a default icon form for most clients, but clients can provide their own if desired, possibly with dynamic rather than static contents. The client is still insulated from management policy, even if it provides the icon: The

manager is responsible for positioning, mapping, and unmapping the icon, and the client is responsible only for displaying the contents.

The icon state is maintained in the server not only to allow clients to provide icons, but to avoid the loss of state if the window manager should terminate abnormally. When a window manager terminates, any windows it has created are destroyed, including icon windows. With knowledge of icons, the server can detect when an icon is destroyed and automatically remap the associated client window. Without this, abnormal termination of the window manager would result in "lost" windows.

9.3 Race Conditions

There are many race conditions that must be dealt with in input and window management because of the asynchronous nature of event handling. For example, if a manager attempts to grab the mouse in response to a press of a button, the mouse-grab request might not reach the server until after the button is released, and intervening mouse events would be missed. Or, if the user clicks on a window to attach the keyboard there and then immediately begins typing, the first few keystrokes might occur before the manager actually responds to the click and the server actually moves the keyboard focus. A final example is a simple interface in which clicking on a window lowers it. Given a stack of three windows, the user might rapidly click twice in the same spot, expecting the top two windows to be lowered. Unless the first click is sent to the manager and the resulting request to lower is processed by the sever before the second click takes place, the event window for the second click will be the same as for the first click, and the manager will lower the first window twice.

A work-around for the last example, used by existing managers, is to ignore the event window reported in most events. Instead, the global mouse coordinates reported in the event are used in a follow-up query request to determine which top-level window now contains that coordinate. However, not all race conditions have acceptable solutions within the current X design. For a general solution it must be possible for the manager to synchronize operations explicitly with event processing in the server. For example, a manager might specify that, at the press of a button, event processing in the server should cease until an explicit acknowledgment is received from the manager.

10. FUTURE

On the basis of critiques from numerous universities and commercial firms, fairly extensive evaluation and redesign of the X protocol have been under way since May 1986. Our desire is to define a "core" protocol that can serve as a standard for window system construction over the next several years. We expect to present the rationale for this new design in the very near future, once it has been validated by at least a preliminary implementation. In this section, we highlight the major protocol changes.

10.1 Resource Allocation

Since the server is responsible for assigning identifiers to resources, each resource allocation currently requires a round-trip time in order to perform. For applications that allocate many resources, this causes a considerable start-up

delay. For example, a multipane menu might consist of dozens of windows, numerous fonts, and several different mouse cursors, leading to a delay of 1 second or longer.

In retrospect, this is the most significant defect in the design of X. To get around these delays, programming interfaces have been augmented to provide "batch mode" operations. If several resources must be created, but there are no interdependencies among the allocation requests, all of the requests are sent in a batch, and then all of the replies are received. This effectively reduces the delay to a single round-trip time.

A better solution to this problem is to make clients generate the identifiers. When the client establishes a connection to the server, it is given a specific subrange from which it can allocate. This change will significantly improve start-up times without affecting applications, as identifiers can be generated inside low-level libraries without changing programming interfaces.

10.2 Transparent Windows

Transparent windows can be used as clipping regions; however, they are unsatisfactory for this purpose because every coordinate in a graphics request must be translated by the client from the "real" window's origin to the transparent window's origin. A better approach to clipping regions is to allow clients to create clipping regions and attach them to all graphics requests. As noted in Section 6, X currently allows a clipping region in the form of a bitmap to be attached to a few graphics requests. Allowing a clipping region, specified either as a bitmap or a list of rectangles, to be attached to all graphics requests provides a more uniform mechanism.

To date transparent windows have been primarily used as inexpensive opaque windows. In the current server implementation, transparent windows can be created and transformed significantly faster than opaque windows. Because of this, transparent windows are often used when opaque windows would otherwise be adequate. We believe a new implementation of the server will improve the performance of opaque windows to the point at which this will no longer be necessary.

With explicit clipping regions added for graphics and the performance advantages of transparent windows reduced, the only remaining use of transparent windows is for input (and cursor) control. Various applications want relatively fine-grained input control, and such control must not affect graphics output. Close control of cursor images and mouse motion events seems particularly important. However, the vast majority of the time control naturally is associated with normal window boundaries, so it would be unwise to divorce input control completely from windows. As such, the new protocol provides "input-only" windows, which act like normal windows for the purposes of input and cursor control, but which cannot be used as a source or destination in graphics requests, and which are completely invisible as far as output is concerned.

10.3 Color

X originally was not designed to deal with direct-color displays. Direct-color displays typically have between 12 and 36 bits per pixel; the pixel value consists of three subfields, which are used as indexes into three independent color maps:

one for red intensities, one for green, and one for blue. Some direct-color displays also have a fourth subfield, sometimes referred to as "z-channel" information, used to control attributes such as blending or chroma keying. We now understand how to incorporate direct-color displays without z-channel information into X in such a way that the differences between direct-color and pseudocolor color maps need not be apparent to the application, yet still allow all of the usual color map tricks to be played.

At present there is only one color map for all applications, and color applications fail when this map gets full. Although dozens of applications typically can be run under X within a single 8-bit pseudocolor map, a single map is clearly unacceptable when dealing with small color maps or with multiple applications (e.g., CAD tools) that need large portions of the color map. The solution is to support multiple virtual color maps, still permitting applications to coexist within any map, but allowing the possibility that not all applications show true color simultaneously. This also matches next-generation displays, which actually support multiple color maps in hardware [39].

10.4 Graphics

Perhaps the biggest mistake in the graphics area was failing to support fonts with kerning (side bearings) [26]. For example, a relatively complete emulation of the Andrew programming interface was built for X, but Andrew applications depend heavily on kerned fonts. There are other deficiencies that will be corrected. For example, large glyph-sets (e.g., Japanese) will be supported, as well as stippling (using a clip mask constructed by tiling a region with a bitmap). The notions of line width, join style, and end style found in PostScript [1] are usually preferred to brush shapes for line drawing and will be supported.

In an attempt to support a wide range of devices, the exact path followed for lines and filled shapes was originally left undefined in X (the class of curve was not even specified). Different devices use slightly different algorithms to draw straight lines, and it seemed better to have high performance with minor variation than to have uniformity with poor performance. Relatively few devices support curve drawing in hardware, but some support it in firmware, and again performance seemed more important than accuracy. In retrospect, however, allowing such device-dependent behavior was a poor decision. The vast majority of applications draw lines aligned on an axis, and speed and precision are not an issue. The applications that do require complex shapes also require predictable results, so precise specifications are important.

A notable feature missing in X is the ability to perform graphics offscreen. The reasons for this are essentially the same as those presented in the discussion of exposures (Section 7). In particular, not all graphics coprocessors can operate on host memory, and emulating such processors can be expensive. However, application builders have demanded this capability, and the demand appears to be sufficient leverage for convincing server implementors to provide the capability. Offscreen graphics will be possible in the new protocol, although the amount of offscreen memory and its performance characteristics may vary widely. In addition, the protocol is being extended to allow the manipulation of both images and windows of varying depths. For example, a server might support depths of 1, 4, 8, 12, and 24 bits. This allows imaging applications to transmit data more

compactly, allows for more efficient memory utilization in the server, and provides a match with next-generation display hardware.

A common debate in graphics systems is whether and where to have state. Should parameters such as logic function, plane mask, source pixel value or tile, tiling origin, font, line width and style, and clipping region be explicit in every request or collected into a state object? The current X protocol is stateless for the following reasons: Both state and stateless programming interfaces can be easily built on top of the protocol; the currently supported graphics requests have just few enough parameters for them to be represented compactly; and the initial set of displays we were interested in (and the implementations we had in mind for them) would not benefit from the addition of state. However, we now believe that a state-based protocol is generally superior, since it handles complex graphics gracefully and allows significantly faster implementations on some displays.

10.5 Management

An obvious interface style presently not supported in X is the ability to use the keyboard for management commands. To allow this, a key-grab mechanism, akin to the button-grab mechanism described in Section 9, will be provided. To allow such styles as using the first button click in a window to attach the keyboard, both button grabs and key grabs have been extended to apply to specific subhierarchies, rather than always to the entire screen. To handle the kinds of race conditions described in Section 9, a general event synchronization mechanism has been incorporated into the grab mechanisms.

To support automatic window management, a manager must be able to intercept certain management requests from clients (such as mapping or moving a window) before they are executed by the server, and to be notified about others (such as unmapping a window) after they are executed. In addition, some managers want to provide uniform title bars and border decorations automatically. To allow this, it is useful to be able to "splice" hierarchies: to move a window from one parent to another. To allow input managers and window managers to be implemented as separate applications, the ability for multiple clients to select events on the same window is being added. For example, both a window manager and an input manager might be interested in the unmapping or destruction of a window.

10.6 Extensibility

The information that input and window managers might desire from applications is quite varied, and it would be a mistake to try to define a fixed set. Similarly, the information paths between applications (e.g., in support of "cut and paste") need to be flexible. To this end, we are adding a LISPish property list [29] mechanism to windows, and the event mechanism is being augmented to provide a simple form of interclient communication.

The new X protocol explicitly continues to avoid certain areas, such as 3-D graphics and antialiasing. However, a general mechanism has been designed to allow extension libraries to be included in a server. The intention is that all servers implement the "core" protocol, but each server can provide arbitrary extensions. If an extension becomes widely accepted by the X community, it can

be adopted as part of the core. Each extension library is assigned a global name, and an application can query the server at run time to determine whether a particular extension is present. Request opcodes and event types are allocated dynamically, so that applications need not be modified to execute in each new environment.

11. SUMMARY

The X Window System provides high-performance, high-level, device-independent graphics. A hierarchy of resizable, overlapping windows allows a wide variety of application and user interfaces to be built easily. Network-transparent access to the display provides an important degree of functional separation, without significantly affecting performance, that is crucial to building applications for a distributed environment. To a reasonable extent, desktop management can be custom-tailored to individual environments, without modifying the base system and typically without affecting applications.

To date, the X design and implementation effort has focused on the base window system, as described in this paper, and on essential applications and programming interfaces. The design of the network protocol and the color allocation mechanism, the design and implementation of device-independent layer of server, and the implementation of several applications and a prototype window manager were carried out by the first author. The design and implementation of the C programming interface, the implementation of major portions of several applications, and the coordination of efforts within Project Athena and Digital Equipment Corporation were carried out by the second author. In addition, many other people from Project Athena, the Laboratory for Computer Science, and institutions outside MIT have contributed software.

Necessary applications, such as window managers and VT100 and Tektronics 4014 terminal emulators, have been created, and numerous existing applications, such as text editors and VLSI layout systems, have been ported to the X environment. Although several different menu packages have been implemented, we are only now beginning to see a rich library of tools (scroll bars, frames, panels, more menus, etc.) for facilitating the rapid construction of high-quality user interfaces. Tool building is taking place at many sites, and several universities are now attempting to unify window systems work with X as a base, so that such tools can be shared.

The use of X has grown far beyond anything we had imagined. Digital has incorporated X into a commercial product, and other manufacturers are following suit. With the appearance of such products and the release of complete X sources on the Berkeley 4.3 UNIX distribution tapes, it is no longer feasible to track all X use and development. Existing applications written in C are known to have been ported to 7 machine architectures of more than 12 manufacturers, and the C server to 6 machine architectures and more than 16 display architectures. In most cases the code is running under UNIX, but other operating systems are also involved. In addition, relatively complete server implementations exist in two LISP dialects. Apart from the portability of the system's design, a large part of this success is due to MIT's decision to distribute X sources without any

licensing restrictions, and the willingness of people in both educational and commercial institutions to contribute code without restrictions.

ACKNOWLEDGMENTS

Our thanks go to the many people who have contributed to the success of X. Particular thanks go to those who have made significant contributions to the nonproprietary implementation: Paul Asente (Stanford University), Scott Bates (Brown University), Mike Braca (Brown), Dave Bundy (Brown), Dave Carver (Digital), Tony Della Fera (Digital), Mike Gancarz (Digital), James Gosling (Sun Microsystems), Doug Mink (Smithsonian Astrophysical Observatory), Bob McNamara (Digital), Ron Newman (MIT), Ram Rao (Digital), Dave Rosenthal (Sun), Dan Stone (Brown), Stephen Sutphen (University of Alberta), and Mark Vandevoorde (MIT).

Special thanks go to Digital Equipment Corporation. A redesign of the protocol and a reimplementation of the server to deal with color and to increase performance were made possible with funding (in the form of hardware) from Digital. To their credit, all of the resulting device-independent code remained the property of MIT.

REFERENCES

1. ADOBE SYSTEMS. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Mass., 1985.
2. APOLLO COMPUTER. *Domain System User's Guide.* Apollo Computer, Chelmsford, Mass., 1985.
3. ASENTE, P. W reference manual. Internal document, Dept. Computer Science, Stanford Univ., Calif., 1984.
4. BALKOVICH, E., LERMAN, S., AND PARMELEE, R. P. Computing in higher education: The Athena experience. *Commun. ACM 28,* 11 (Nov. 1985), 1214–1224.
5. CHERITON, D. The V kernel: A software base for distributed systems. *IEEE Softw. 1,* 2 (Apr. 1984), 19–42.
6. COHEN, D. On holy wars and a plea for peace. *Computer 14,* 10 (Oct. 1981), 48–54.
7. DIGITAL EQUIPMENT CORP. *VCB02 Video Subsystem Technical Manual.* Educational Services, Digital Equipment Corporation, Bedford, Mass., 1986.
8. GANCARZ, M. UWM: A user interface for X windows. In *Summer Conference Proceedings* (Atlanta, Ga., June 10–13). USENIX Association, 1986, pp. 429–440.
9. GETTYS, J. Problems implementing window systems in Unix. In *Winter Conference Proceedings* (Denver, Colo., Jan. 15–17). USENIX Association, 1986, pp. 89–97.
10. GOSLING, J., AND ROSENTHAL, D. A window-manager for bitmapped displays and Unix. In *Methodology of Window-Managers,* F. R. A. Hopgood et al., Eds. Springer-Verlag, New York, 1986.
11. HAWLEY, M. J., AND LEFFLER, S. J. Windows for Unix at Lucasfilm. In *Summer Conference Proceedings* (Portland, Oreg., June 11–14). USENIX Association, 1985, pp. 393–406.
12. INTERNATIONAL STANDARDS ORGANIZATION. Information processing: Graphical kernel system (GKS)—Functional description. Rep. DIS 7942, International Organization for Standardization, Geneva, Switzerland, 1982.
13. LANTZ, K. A., AND NOWICKI, W. I. Structured graphics for distributed systems. *ACM Trans. Graph. 3,* 1 (Jan. 1984), 23–51.
14. LEVY, H. VAXstation: A general-purpose raster graphics architecture. *ACM Trans. Graph. 3,* 1 (Jan. 1984), 70–83.
15. LIPKIE, D. E., EVANS, S. R., NEWLIN, J. K., AND WEISSMAN, R. L. Star graphics: An object-oriented implementation. *Comput. Graph. 16,* 3 (July 1982), 115–124.
16. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst. 5,* 3 (July 1983), 381–404.

17. MCKEE, L. *MC-WINDOWS Programming Manual, Revision A.* Massachusetts Computer Corporation, Westford, Mass., 1985.
18. MICROSOFT CORP. *Microsoft Windows: Programmer's Guide.* Microsoft Corporation, Redmond, Wash., 1985.
19. MOON, D. Chaosnet. AI Memo 628, Artificial Intelligence Laboratory, MIT, Cambridge, Mass., June 1981.
20. MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S. H., AND DONELSON SMITH, F. Andrew: A distributed personal computing environment. *Commun. ACM 29,* 3 (Mar. 1986), 184–201.
21. MYERS, B. Issues in window management design and implementation. In *Methodology of Window-Managers,* F. R. A Hopgood et al., Eds. Springer-Verlag, New York, 1986.
22. NOWICKI, W. Partitioning of function in a distributed graphics system. Ph.D. dissertation, Dept. Computer Science, Stanford Univ., Calif., 1985.
23. PIKE, R. The Blit: A multiplexed graphics terminal. *AT&T Bell Lab. Tech. J. 63,* 8 (Oct. 1984), 1607–1631.
24. POSTEL, J. Transmission control protocol. Rep. RFC 793, USC/Information Sciences Institute, Marina del Rey, Calif., Sept. 1981.
25. RHODES, R., HAEBERLI, P, AND HICKMAN, K. Mex—A window manager for the IRIS. In *Summer Conference Proceedings* (Portland, Oreg., June 11–14). USENIX Association, 1985, pp. 381–392.
26. ROSENTHAL, D. Window system implementations. USENIX Association, 1986. (Course notes for *Winter Conference,* Denver.)
27. SMITH, D. C., IRBY, C., KIMBALL, R., AND HARSLEM, E. The Star user interface: An overview. In *Proceedings of the 1982 National Computer Conference* (Houston, Tex., June 7–10). AFIPS Press, Reston, Va., 1982, pp. 515–528.
28. STALLMAN, R., MOON, D., AND WEINREB, D. *Lisp Machine Window System Manual.* MIT Artificial Intelligence Laboratory, Cambridge, Mass., Aug. 1983.
29. STEELE, G. L. *Common Lisp: The Language.* Digital Press, Bedford, Mass., 1984.
30. SUN MICROSYSTEMS. *Programmer's Reference Manual for SunWindows.* Sun Microsystems, Mountain View, Calif., 1985.
31. SUN MICROSYSTEMS. *NeWS Preliminary Technical Overview.* Sun Microsystems, Mountain View, Calif., 1986.
32. SWEET, R. Mesa programming environment. *ACM SIGPLAN Not. 20,* 7 (July 1985), 216–229.
33. SWEETMAN, D. A modular window system for Unix. In *Methodology of Window-Managers,* F. R. A. Hopgood et al., Eds. Springer-Verlag, New York, 1986.
34. SYMBOLICS. *Programming the User Interface.* Symbolics, Cambridge, Mass., 1986.
35. TEITELMAN, W. The Cedar programming environment: A midterm report and examination. Rep. CSL 83-11, Xerox PARC, Palo Alto, Calif., June 1984.
36. TRAMMEL, R. D. A capability based hierarchic architecture for Unix window management. In *Summer Conference Proceedings* (Portland, Oreg., June 11–14). USENIX Association, 1985, pp. 373–379.
37. WARNOCK, J., AND WYATT, D. K. A device independent graphics imaging model for use with raster devices. *Comput. Graph. 16,* 3 (July 1982), 313–319.
38. WECKER, S. DNA: The digital network architecture. *IEEE Trans. Commun. COM-28,* 4 (Apr. 1980), 510–526.
39. WILKES, A. J., SINGER, D. W., GIBBONS, J. J., KING, T. R., ROBINSON, P., AND WISEMAN, N. E. The Rainbow workstation. *Comput. J. 27,* 2 (May 1984), 112–120.

Received July 1986; revised October 1986; accepted October 1986

APPENDIX G

The Role of Balloon Help

David K. Farkas
Department of Technical Communication
College of Engineering
University of Washington

Abstract. *Balloon Help, which is becoming standard in the Macintosh world, enables the user to display brief annotations of interface objects by passing the pointer (cursor) over those objects. This investigation explains the operation of Balloon Help, presents the theoretical and empirical rationale for Balloon Help, assesses its value in supporting both exploration of an interface and task-focused behavior, considers its relationship with other forms of help, and evaluates some possible modifications of Balloon Help. Balloon Help is viewed as a successful implementation of minimalist principles that nevertheless needs to be supplemented by other forms of documentation.*

Balloon Help was introduced into the Macintosh operating system with the System 7 release. Balloon Help was used by Apple in documenting System 7, and Apple strongly supports its use by all developers of software for the Macintosh. A great many developers are incorporating Balloon Help as they introduce new products and recode their existing products for System 7, and so Balloon Help will very likely become standard in the Mac world (Gassée, 1991).

In this study, I analyze and assess Balloon Help. Specifically, I

- explain its operation,
- present it as a form of interface annotation incorporating minimalist principles,
- assess its effectiveness when used both for initial familiarization with a product and for accomplishing tasks,
- consider its relationship to and potential duplication of other forms of online help, and
- evaluate some possible enhancements of Balloon Help.

Balloon Help has received generally favorable commentary in the trade press (Swaine, 1990; Matthies, 1991; Poole, 1991; Davis, 1991); it is applauded for providing users with quick, convenient access to help information. But the reviews are not consistently positive. One commentator calls Balloon Help "little more than a gimmick" (Reed, 1991), while another refers to it as "something my 5-year old child needed occasionally" (Levitan, 1991). In my own

experience, Balloon Help has left many experienced Mac users unimpressed. It is not, after all, a stunning technological advance: It employs no AI technology, interactive dialog with the user, multimedia, or sophisticated hypertext linking. Much flashier online aids have appeared of late. Furthermore, if a help system is measured by the amount of information it delivers to the user, Balloon Help is certainly less than impressive. Despite all the things it is not, Balloon Help, I maintain, is effective in a variety of situations. Simple and undramatic, it is an instance of appropriate technology in the world of online assistance.

How Balloon Help Works

Users enter the Balloon Help mode by selecting the Show Balloons command from the Help menu on the Macintosh menu bar. They exit this mode by means of the Hide Balloons command on the same menu. Once in the Balloon Help mode, many interface objects on the screen are "hot." That is, they will display small "balloons" (see Figure 1) containing brief help messages when the user moves the pointer (cursor) over them. These balloons, which are named after the balloons used in comic book dialog, appear at the location of the hot spot. Each balloon has a "tip" that points precisely at the hot interface object. Balloons are parsimonious in design: there are no buttons to click, no scroll bars, no title area. The dimensions of the balloon are barely larger than the space required for the balloon message.

Which spots will generate balloons? This depends on the software developer. Balloons for the Title bar, Close box, and other unvarying Macintosh interface objects are provided by System 7. But software developers can put balloons almost anywhere, and the process is relatively easy, especially with the BalloonWriter utility. Even temporary interface objects like handles for graphics can trigger balloons. Furthermore, a different message can be written for the same spot when it is in a different condition. That is, an option button (radio button) may display a different message when it is selected, unselected or unavailable for selection (dimmed). Other than this moderate degree of context sensitivity, the display of balloons is unrelated to deeper-level changes in the system's state or the actions of the user; Balloon Help is not intelligent online assistance. For both technical and communicative reasons balloon messages must be short, and although graphics can be

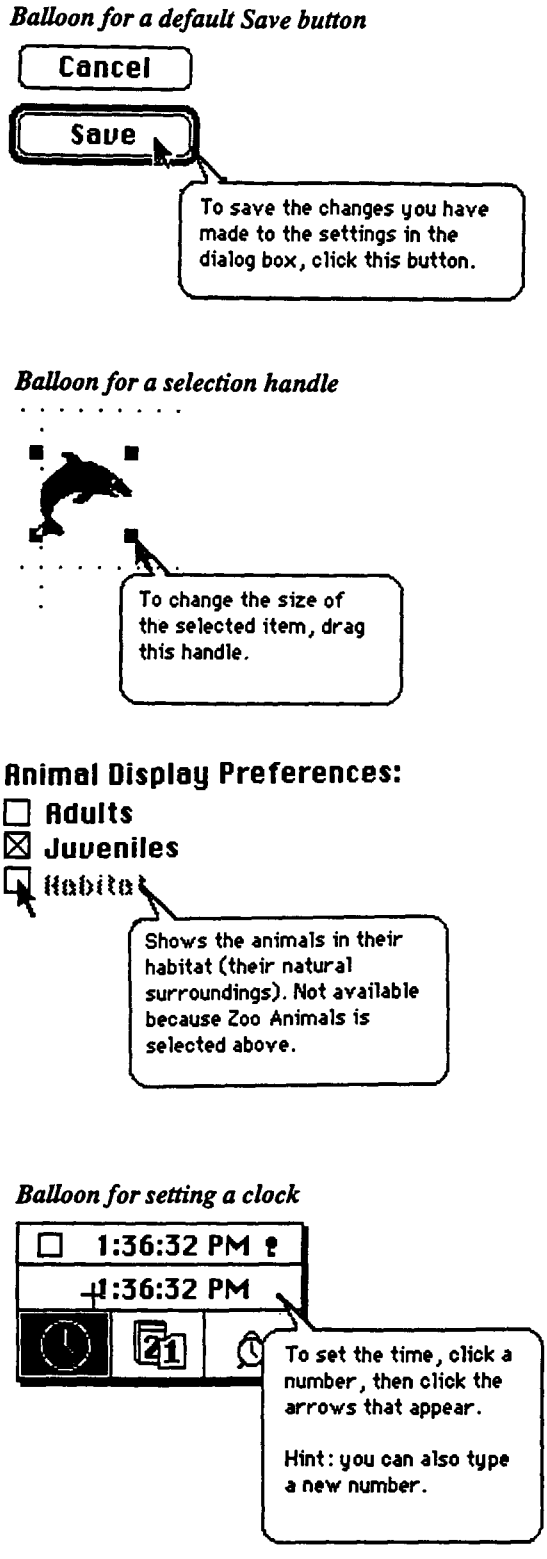


Figure 1. Several examples of Balloon Help on display.

placed in balloons, this is rarely done. Typically, the messages explain the purpose of the interface object (*Apple Publications Style Guide*, 1991).

A balloon is dismissed as soon as the pointer leaves the hot spot; thus only one balloon at a time is displayed. Even so, an often-cited problem is "balloon barrage." Users are distracted by numerous balloons that appear unintentionally as the user moves the pointer toward the next object of interest. To limit the number of unintentional balloons, the pointer must pause for 1/10 second over a hot spot before a balloon is triggered (Matthies, 1991). Consequently, if the user moves the pointer decisively from one location to the next, unintended balloons will not appear. Computer users, however, do not necessarily move the pointer decisively, especially when they are looking for information, and so balloon barrage remains an issue.

One solution for balloon barrage has been offered by the developers of "init" utilities, such as Helium, that enable the user to toggle in and out of Balloon Help mode from the keyboard. In the case of Helium, Balloon Help is active only while a key combination is held down. This quick-toggle feature permits the highly selective display of balloons, but eliminates the automatic, effortless quality of the original implementation. Summing up the salient features of Balloon Help, we can say:

1. Balloon Help is spot-triggered. Unlike many forms of context-sensitive help, a developer can trigger help information from almost any pixel on the screen.
2. Balloon Help is spot-displayed. That is, in contrast to many forms of context-sensitive help, balloons appear right next to the object that triggered them. Because of spot-display and also because of the balloon tip and the small size of balloons, there is a close association in the user's mind between a balloon and the object that triggers it.
3. Balloon Help has some awareness of the system state and separate balloons can be written for the same object in different states.

One way is to point out a limitation of the now-dominant graphical-user interface (GUI): the numerous graphical controls and icons they contain are not completely intuitive and self-disclosing, and there is rarely enough screen "real estate" to explain everything users need to know about the system. New users of a GUI do not reliably infer the function of such standard objects as scroll bars and zoom boxes, and even users who are experienced with a particular GUI cannot predictably infer the function of the various application-specific icons and other graphical objects they encounter in an unfamiliar product. In addition to mysterious graphical objects, graphical-user interfaces contain many text labels, such as command names and labeled check boxes and option buttons. Because of space constraints, these labels are often too brief to be meaningful. Non-graphical interfaces face much the same problem, but there is likely to be a higher proportion of brief text labels to graphical objects. Balloons, then, can be thought of as interface annotations, elaborative comments. But whereas permanently displayed screen annotations, "persistent help" in Kearsley's terms (1988), would hopelessly clutter the screen if they were placed everywhere the user could benefit from them, balloon annotations appear when they are needed and disappear when they are not.

A second rationale can be drawn from the theoretical and empirical work conducted for IBM by John Carroll (Carroll, Smith-Kerker, Ford, and Mazur-Rimetz, 1987-88; Carroll and Rosson, 1987; Carroll, 1990). This work defined and popularized the concept of minimalist documentation. Carroll made the important observation that computer users are impatient and highly curious and that, rather than reading extended documentation, they want to begin immediately working with the product (Carroll, Smith-Kerker, Ford, and Mazur-Rimetz, 1987-88; Carroll and Rosson, 1987). Carroll also observed that, even while they are just starting to learn a system, users want to get actual work done, and—again—that users prefer to bypass documentation or use the briefest possible documentation as they accomplish their work (Carroll and Rosson, 1987). The minimal manual was Carroll's primary effort to satisfy these desires of computer users (Carroll, Smith-Kerker, Ford, and Mazur-Rimetz, 1987-88). Balloon Help can be seen as an online implementation of the original minimalist idea. Users forego introductions, conceptual overviews, any instructional curriculum, and complete procedures for quick access to explanations and hints that will support their own explorations and task-focused efforts with the software.

A third rationale comes from empirical research on the behavior of computer users conducted by Sellen and Nicol (1990) as part of Apple Computer's ongoing research in the area of online assistance. Sellen and Nicol report that computer users formulate the following five kinds of questions when trying to learn an unfamiliar software product.

1. Goal oriented—What kinds of things can I do with this program?
2. Descriptive—What is this? What does this do?
3. Procedural—How do I do this?
4. Interpretive—Why did that happen? What does this mean?
5. Navigational—Where am I?

Sellen and Nicol then briefly describe a research prototype quite similar to Balloon Help as a means of providing descriptive information. Furthermore, they note that descriptive information is needed in two different modes of user behavior: The first is an exploratory mode in which users are becoming familiar with the interface; the second is a task-focused mode in which users need explanations of objects when they are focused on accomplishing actual tasks. Their work, therefore, like Carroll's provides a general rationale for Balloon Help and points more specifically to two roles it can play in supporting users. We will now turn from the general rationale for Balloon Help to an assessment of its roles in the full documentation set, and we will begin by considering Balloon Help both as a means of exploring an interface and learning how to perform tasks.

Balloon Help For Familiarization

Many forms of documentation, both print and online, can help the user explore and gain an overall familiarity with a new product. Both tutorials and user's guides can serve this end. Online demos (or "tours") are intended specifically for familiarization. Balloon Help, however, is ideal in supporting the user's initial exploration of a product or a part of the product the user has not yet examined. First, Balloon Help provides user-directed exploration in which the user's curiosity rather than some instructional curriculum is satisfied. Also important is the speed at which balloons are displayed. Information comes a mere mouse

flick after the user's attention has been drawn to an object. Not even the most helpful human tutor can respond as quickly to a user's interest in some part of the interface—and the discourse of a human tutor is harder to turn off than a balloon.

Spot display also contributes to initial familiarization. As Carroll has noted, one problem users face learning new systems is correctly focusing their attention. They often miss important messages and cues (Carroll, 1990, p. 34). But the close association of the balloon message with the object of interest eliminates this problem.

One limitation of Balloon Help is the necessary brevity of balloon text. Brevity, however, is very appropriate for initial exploration. Furthermore, in Balloon Help every word counts. Manuals and conventional help screens need to include verbal descriptions or else screen representations to show the user where on the screen to look for the object being explained. This, of course, is unnecessary in Balloon Help, where the interface serves as its own graphic. Also, there is often no need to state the action the user must perform to execute a step in a procedure. The close association of the balloon with a particular interface object—a menu choice, a button, a checkbox, etc.—in many situations adequately implies the appropriate action.

Finally, although balloons are not a form of intelligent help, the fact that separate balloons can be written for objects that are selected, unselected, and unavailable for selection adds considerable value. Users particularly want an explanation when the object they are interested in is unavailable.

There is at least informal empirical support for the value of Balloon Help in initial exploration. Michael Hancheroff (1991), who provides support to users at the University of Washington Microcomputer Showroom, observed that users unfamiliar with System 7 and with Macintosh applications offering Balloon Help typically explore the interface with Balloon Help for a short period of time, an hour at most, and then turn it off. Also, Marshall McClintock (1992) reports observations on Balloon Help that were incidental to other usability tests he conducted at Microsoft. McClintock found that sessions with Balloon Help, though very brief, can "have a dramatic influence," especially in providing users with the background to make good use of other forms of help. Given that users basically dislike documentation of all kinds, the brevity of Balloon Help sessions, if the sessions are productive, is in my judgment no ground for criticism.

Balloon Help for Tasks

While important, familiarization is only one part of a user's life history with a software product. Users, as noted above, are curious and wish to explore, but they also have a very strong urge to accomplish actual work. How well, then, does Balloon Help support users when they have completed initial familiarization and are seeking online help in support of real tasks?

Finding the Right Balloon

Information access is not an issue when users are exploring an interface to gain familiarity. The user sees an object, wonders about it, moves the pointer, and views a balloon. But once the user has committed to trying to accomplish a particular task, information access becomes paramount: users must identify the appropriate interface objects before they can access the relevant balloon. Balloon Help, therefore, requires the user to draw inferfences from the interface. Carroll applauds documentation that encourages this kind of active learning as well as documentation that keeps the user's focus on the working interface rather than on pages of a manual or windows of help information.

The success of this problem-solving activity, however, depends both on a particular user's skill at inferring and the quality of the interface. If the key control for the desired task is buried four levels deep in the interface or is placed under an unlikely menu, the user might never find the requisite control and its balloon. There are also procedures that are not associated with any particular part of the interface, leaving the help writer with no good place to associate a balloon. On the other hand, if the interface is well designed, information access via Balloon Help is likely to be fast and accurate. A prototypical instance is the user who finds the command that seems to match the task goal, uses the balloon to confirm that choice, displays the dialog box for that command, and then uses the dialog box balloons to provide convenient capsule explanations of the dialog box options.

Following the Procedure

Balloons are necessarily brief. As Sellen and Nicol point out, balloons often describe the function of an interface object rather than present a series of steps that will encompass the complete task. In the case of dialog box options, this is no limitation, because the dialog box option is, in effect, a self-contained mini-procedure that enables a user to complete a task in the exact manner the user

desires (e.g., printing, but printing only a portion of the document).

If, on the other hand, the user selects a block of text, displays the balloon for the Copy command, and reads that the copy command "copies the selected text and graphics to the Clipboard," the user must be able to complete the task (pasting the selected text or graphic back into the document) from other knowledge of the system. Alternatively, the help writer may write a longer balloon that includes an explanation of pasting, thus relieving the user of this burden. The sample balloon for setting the time is interesting and impressive because in only 21 words it packs a purpose statement (to set the time), and three action steps (clicking a number, clicking the arrows, and the alternative method, typing a number), along with a feedback step (arrows will appear).

Many products, however, require much lengthier and conceptually more complex procedures in which several steps are decision points (conditionals) for which guidelines must be provided. In these instances, the inability of balloons to provide more than the briefest conceptual information and feedback information becomes a limitation. Also, many complex procedures require users to operate controls located in disparate parts of the interface. Few users will be willing to successively consult a series of balloons as they carry out a single procedure.

Clearly, then, both in terms of information access and presentation, there will be products and portions of products in which the practical limits of Balloon Help are exceeded. Both Sellen and Nicol and the *Apple Publications Style Guide* acknowledge this fact. On the other hand, Balloon Help is an excellent means of providing familiarity and supports task-focused behavior over a broad range of product functionality.

Complementing Balloon Help

If Balloon Help does not fully support task-focused behavior, a good means of complementing Balloon Help is not far to seek. One of the most prevalent forms of help consists of windows or panels of help information accessed by a hierarchy of descriptive phrases. The user scans a menu or some other listing of top-level entries and then navigates down into a hierarchy of more specific entries until finding the title of the desired procedure. (Other

hierarchies can be devised for commands, keyboard shortcuts, etc.) Accessing windows or panels of procedural information in this way is comparable to using the table of contents in a printed user's guide. Constructing an effective procedure hierarchy requires a systematic analysis of the tasks the user might want to perform and a mapping of these tasks to the functionality of the product. But if this is done correctly, the access is relatively immune to quirks in the interface, and supports users who do not want to explore an interface and infer which objects support which tasks. These users only deal with the interface when they follow instructions for carrying out a procedure.

Another traditional form of access is the keyword list or online index, the online equivalents of the traditional back-of-book index. Here the help writer complies an extensive alphabetical listing of words and phrases that are meant to correspond to the phrases that users are likely to formulate to represent their goals. In both cases, the user does not find help information from the working interface, but rather consults listings of task-oriented phrases devised by the help writer. So, a good complement to the interface-based access provided by Balloon Help (and various other forms of context-sensitive help) is its diametric opposite, what we can call "phrase-based" access to help information.

Apart from information access, another reason why phrased-based access is a good complement to Balloon Help is information presentation. In almost all implementations, phrase-based access provides much more complete help information than does Balloon Help. Typically, the user is shown a full window or panel from a library of help topics, and this window (or panel) can offer scrolling or paging through the help topic as well as links to other help topics in the help library provided by a browse sequence, hypertext jumps, and pop-up definitions. Also, because these windows are not associated with particular interface objects and, in the better implementations, remain on the screen while the user works with the product, they are well suited for documenting procedures that involve disparate parts of the interface.

Balloon Help and System Prompts: Is There Duplication?

Although phrase-based access to detailed help information complements Balloon Help, there might well be significant duplication of function if a software product included both Balloon Help and some other form of help optimized to display brief help messages.

There is a form of help, in fact, which has been implemented along with Balloon Help in certain products, which in some respects resembles Balloon Help, and which might well be perceived as duplication of Balloon Help. This is help in the form of system prompts. It is worthwhile, therefore, to clarify the relationship between system prompts and balloons and to demonstrate that any duplication is incidental and a kind of historical anomaly.

Many computer products offer system-initiated messages of various kinds. These include error messages, alerts of important actions (Do you wish to overwrite the file: Lovenote), progress and completion messages (Backing up the file: Lovenote . . . Backup completed), and prompts for the next user action.

Error messages and alert messages usually appear prominently on the screen and require some explicit action before the user can resume normal operations with the software. Other system messages, including prompts, typically appear in a small message area located at the bottom or some other margin of the screen and do not require any explicit response. The user simply continues working with the product and can either heed or ignore this information.

In current systems, prompt messages are sometimes similar and even identical to balloon messages. Sun Microsystem's *Open Look UI Style Guide* (1989), for example, suggests the following prompt when the user has selected the rectangle tool from a drawing palette: "Position pointer then drag—Rectangle Tool." A balloon for the rectangle tool might be quite similar. Moreover, in some Microsoft products, certain balloons are similar or identical to prompts that appear in the status line located at the bottom of the screen. If a software product offers two forms of help that might display the same message, is there a significant overlap in function?

One difference, of course, is that prompts are not spot-displayed. Furthermore, whereas balloons are sensitive to the position of the pointer over an object, prompts require the object to be selected or given some explicit focus. This difference has the practical result of limiting the range of objects for which prompts can be written and in making access to prompts slower than access to balloons. But this difference also points to a more fundamental difference between Balloon Help and system prompts, the difference in their essential nature and ultimate evolution. Balloons are annotations and explain the purpose of interface objects. Prompts are directive in nature; they reflect the system's record of the user's recent actions and best guess as to the user's current intentions. In intent, they are not explanations of interface objects but explicit instructions for what to do next.

Currently, prompts can provide explicit instructions only in highly restricted or highly structured domains such as automated bank tellers, simple e-mail systems, logon procedures for mainframes, and data-entry screens. In more complex domains, however, it is often impossible to effectively track and anticipate user actions, and so help writers often can do no more than write balloon-like annotations explaining the function of the most recently selected object. The prompt, then, becomes no more than a hint and a somewhat inferior form of Balloon Help.

But despite the current limitations of prompts in complex domains, computers will achieve intelligent prompting. They will track more user actions, make better inferences about these actions, query users for clarification of their intentions, and offer more detailed advice that can include conditional instructions (If you are trying to do A, then....) Thus, as prompting becomes more intelligent, there will be increasing divergence in the nature of prompt messages and balloon messages.

Modifying Balloon Help

What is the potential for modifying Balloon Help and for creating new, more refined help engines on the general model of Balloon Help? What kinds of changes are worthwhile enhancements? Should changes be implemented that alter the fundamental character of Balloon Help?

A common thread running through all these potential modifications is the issue of added complexity, either in the way in which the user operates Balloon Help or in the nature of the help display. Complexity is the bane of help systems, and so careful thought and usability testing are necessary to confirm the value of any potential modification.

Quick Toggling

The prevalence of Helium indicates user interest in a quick-toggle feature that permits balloons to be accessed deliberately. The choice between system-initiated and user-initiated display of balloons is highly individual, but a means of preventing unwanted balloons would be particularly valuable for users engaged in task-focused activity.

This is because task-focused activity, much more than familiarization, requires problem-solving and other deeper-level mental processes, and so balloon barrage can be much more distracting. The proliferation of balloons is competing for scarce processing resources (Navon and Miller, 1987). I have personally seen several users turn off Balloon Help as they made the transition from familiarization to trying to accomplish a new task. Had Helium been installed, they might well have continued to use Balloon Help. It therefore seems that with some sort of Helium-like quick toggle, a major impediment for using Balloon Help to support task-focused behavior is removed.

Filtering System-Level Balloons

A problem related to balloon barrage is that users who have learned the basics of the Macintosh interface soon tire of repeatedly viewing the balloons for standard and very familiar objects such as the title bar, scroll bar, and inactive windows. A possible modification of Balloon Help, therefore, is a filtering option that would eliminate the underlying System 7 "beginner's balloons" and retain only balloons specific to the application being used. The principle of "layering" information for different users is central to documentation, and this modification is an implementation of this principle.

Revealing the Area Triggering a Balloon

Users can benefit from an immediate visual cue indicating the number of interface objects a particular balloon pertains to. For example, the use of highlighting could more strongly distinguish an instance in which one balloon pertains collectively to three related option buttons from an instance in which three related option buttons have three separate balloons. This concept was implemented in the

precursor to Balloon Help that appears in some HyperCard 2.0 help panels. When the balloon is displayed, the entire region that triggered the balloon is highlighted until the user moves the pointer out of this region and dismisses the balloon (Figure 2). Although this feature adds visual complexity to balloon display, it might be worth implementing.

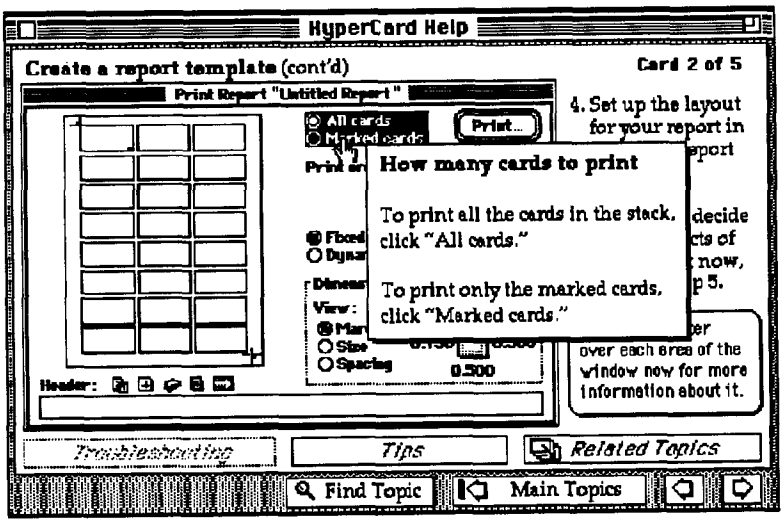


Figure 2. HyperCard 2.0 help screen showing highlight for the area that triggered a "balloon."

Links to the Standard Help Library

Many software products offer context-sensitive access to the standard help library normally accessed by topic hierarchies or keywords. In one implementation the user selects a particular interface object and presses a key; in another, the user turns the pointer into a special help pointer and then selects an object. This form of help supports interface-oriented problem-solving, somewhat as Balloon Help does, but provides detailed information rather than brief balloon messages.

A possible enhancement to Balloon Help is to provide rapid access from any balloon to the most appropriate screen in the standard help library. Balloons might have their own hot spots (a bit tricky to implement) or the F1 key could be used. The help windows in Sun's OpenWindows Spot Help includes a button that brings up the standard help reference, Help Viewer, displaying a list of topics generally related to the topic of the Spot Help window. The *Open Look UI Style Guide* (see Figure 3) suggests a more elaborate variation in which the help information window contains three buttons for accessing more help information.

Graphics and Multimedia

At a time when multimedia help is appearing, Balloon Help does not even utilize graphics. Should something be done? In many instances, Balloon Help does quite well without graphics. The most common form of graphic in computer documentation is the screen representation that shows a specific portion of the interface to the user. Balloon Help uses its own association with the relevant object as a very adequate substitute for screen representations. While animated documentation is not always the best means of providing help information (Palmiter, Elkerton, and Baggett, 1991), animation is often highly desirable for explaining processes and other documentation tasks. A company called Motion Works has developed a product that software developers can use to create special balloons containing brief animated sequences.

More "Intelligent" Balloons

An interesting issue is the desirability of providing greater context sensitivity and even "intelligence" for Balloon Help. There are clearly benefits in adding greater context sensitivity to Balloon Help. For example, in a word processing program, the command for adding footnotes might trigger separate balloons depending on whether footnotes had already been created for the document. Going further, the way in which footnotes had been added to the document could dictate the nature of the balloon message. But even if greater context sensitivity is added to Balloon Help, its fundamental character should not be changed. It should remain annotative and descriptive, an aid to users as they figure out what to do, and not a means of providing (or trying to provide) explicit directions for the user's next action. Explicit directions are the natural evolution of prompts and other forms of help, such as alerts, which represent the best advice of the system as a whole and are not closely associated with specific objects on the interface. Furthermore, given limited resources, the large

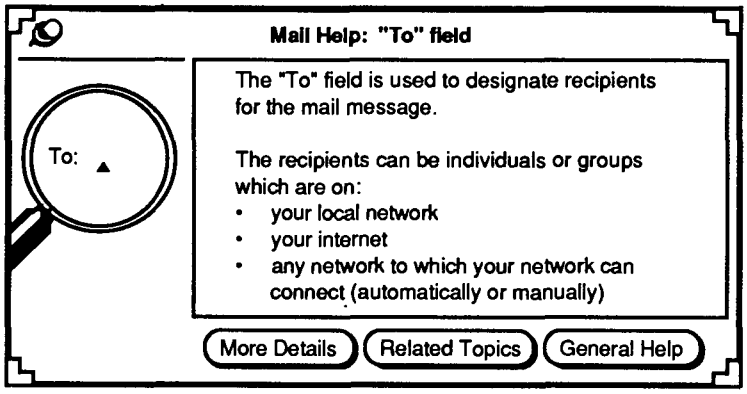


Figure 3. Buttons for immediate access to detailed help information.

investment entailed in developing intelligent help is probably best directed toward system prompts rather than Balloon Help.

Conclusion

This investigation of Balloon Help mentioned three broad families of help: context-sensitive help, help that uses phrased-based access, and help which takes the form of system prompts which track the user's interactions with the system and which aspire beyond context sensitivity toward true intelligence.

Balloon Help is a form of context-sensitive help featuring spot triggering of balloons, spot display of balloons, and balloons optimized for brief messages. Also, whereas most context-sensitive help is user initiated, Balloon Help, with the addition of Helium, can operate in either system-initiated mode (ideal for familiarization) and the more deliberate user-initiated mode (ideal for task-focused learning).

Balloon Help is brief and instantly available, and asks users to keep their attention on the interface and engage in inferential learning. Thus it broadly follows John Carroll's minimalist program, and is, in fact, a successful implementation of minimalism.

Balloon Help, nonetheless, has significant limitations, particularly in supporting long and complex procedures. For this reason, and because users should not be required to rely on the interface to find documentation for the tasks they want to accomplish, Balloon Help should not be the sole piece of documentation or even the sole piece of on-line documentation for a product.

Despite the mixed reception it has received, Balloon Help should have a bright future. The now-dominant graphical-user interfaces seem to sport ever more cryptic graphical objects, and commands have ever more options, all represented in dialog boxes (and other places) in very terse form. All this needs explication.

Another trend favors Balloon Help. As graphical interfaces become ever more prevalent and more standardized, users are becoming more familiar with such basic GUI operations as pulling down menus, clicking buttons, and typing into text boxes. Greater numbers of users, one might

reason, will become impatient with step-by-step procedures that incorporate descriptions of these actions. Brief purpose-oriented statements, the ideal content of balloons, may be increasingly favored, and more users may migrate to Balloon Help from more conventional forms of documentation.

References

Apple Publications Style Guide. (Fall 1991). Cupertino, CA: Apple Computer, Inc.

BalloonWriter User's Guide. (1991). Cupertino, CA: Apple Computer, Inc.

Carroll, J.M. (1990). The Nürnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill, Cambridge, MA: MIT Press.

Carroll, J.M., and Rosson, M.B. (1987). The paradox of the active user. In J.M. Carroll (ed.), Interfacing Thought: Cognitive Aspects of Human Computer Interaction , pp. 80-111. Cambridge, MA: MIT Press/Bradford Books.

Carroll, J.M., Smith-Kerker, P.A., Ford, J.R., and Mazur-Rimetz, S.A. (1987-88). The minimal manual. Human Computer Interaction. 3, pp. 123-153.

Davis, F. (May 20, 1991). Operating systems are evolutionary, not revolutionary. PC Week, 8, p. 165

Gassée, J.L. (August 6, 1991). System 7 getting pledge of allegiance. MacWeek 5, p. 64.

Hancheroff, M.C. (October 25, 1991). Personal Communication.

Levitan, A. (October 1991). A tale of two operating systems. Computer Shopper, 11, pp. 137 & 148.

Kearsley, G. (1988). Online Help Systems: Design and Implementation. Norwood, NJ: Ablex.

Matthies, K.W.G. (1991). Balloon Help takes off. MacUser, December 1991, pp. 241-48.

McClintock, M. (January 17, 1992). Personal Communication.

Navon, D., and Miller, J. (1987). Role of outcome conflict in dual-task interference. Journal of Experimental Psychology: Human Perception and Performance, 12, pp. 435-48.

Open Look UI Style Guide. (1989). Mountain View, CA: Sun Microsystems, Chapters 11 & 12.

Palmiter, S., Elkerton, J., and Baggett, P. (1991). Animated demonstrations vs. written instructions for procedural tasks: a preliminary investigation. International Journal of Man-Machine Studies, 34, pp. 678-701.

Poole, L. (July 1991). Confessions of a System 7 user. Mac World, 8, pp. 195-201.

Reed, S. (August 1991). Apple scouts ahead with System 7. PC Computing, 4, pp. 40-42.

Sellen, A., and Nicol, A. (1990). Building user-centered on-line help. In B. Laurel (ed.) The Art of Human-Computer Interface Design, pp. 143-153. Reading, MA: Addison-Wesley.

Swaine, M. (November 1990). System 7.0 watch: the read balloon. MacUser, 6, p. 244.

Thanks to Jean Farkas, Mark Hancheroff, Linda and David Leonard, Marshall McClintock, Maria Staaf, Kent Sullivan, and Jon Wiederspan.

APPENDIX H

NEW AGE

An Introduction to CLIENT/SERVER COMPUTING

Subhash Chandra Yadav • Sanjay Kumar Singh

NEW AGE INTERNATIONAL PUBLISHERS

An Introduction to CLIENT/SERVER COMPUTING

This page intentionally left blank

An Introduction to CLIENT/SERVER COMPUTING

Subhash Chandra Yadav
M.Sc., M.C.A. and M.Phil. (Computer Science)
Reader
Department of Computer Applications
Rajarshi School of Management and Technology
U.P. College Campus
Varanasi, (U.P.)

Sanjay Kumar Singh
Ph.D. (Computer Science and Engineering)
Reader
Department of Computer Engineering
Institute of Technology,
B.H.U., Varanasi, (U.P.)

PUBLISHING FOR ONE WORLD
NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS
New Delhi • Bangalore • Chennai • Cochin • Guwahati • Hyderabad
Jalandhar • Kolkata • Lucknow • Mumbai • Ranchi
Visit us at **www.newagepublishers.com**

Copyright © 2009, New Age International (P) Ltd., Publishers
Published by New Age International (P) Ltd., Publishers

All rights reserved.

No part of this ebook may be reproduced in any form, by photostat, microfilm, xerography, or any other means, or incorporated into any information retrieval system, electronic or mechanical, without the written permission of the publisher.
All inquiries should be emailed to ***rights@newagepublishers.com***

ISBN (13) : 978-81-224-2861-2

PUBLISHING FOR ONE WORLD

NEW AGE INTERNATIONAL (P) LIMITED, PUBLISHERS
4835/24, Ansari Road, Daryaganj, New Delhi - 110002
Visit us at **www.newagepublishers.com**

Preface

In recent years there have been significant advances in the development of high performance personal computer and networks. There is now an identifiable trend in industry toward downsizing that is replacing expensive mainframe computers with more cost-effective networks of personal computer that achieve the same or even better results. This trend has given rise to the architecture of the Client/Server Computing.

The term Client/Server was first used in the 1980s in reference to personal computers on a network. The actual Client/Server model started gaining acceptance in the late 1980s. The term Client/Server is used to describe a computing model for the development of computerized systems. This model is based on the distribution of functions between two types of independent and autonomous entities: Server and Client. A Client is any process that request specific services from server processes. A Server is process that provides requested services for Clients. Or in other words, we can say "A client is defined as a requester of services and a server is defined as the provider of services." A single machine can be both a client and a server depending on the software configuration. Client and Server processes can reside in same computer or in different computers linked by a network.

In general, Client/Server is a system. It is not just hardware or software. It is not necessarily a program that comes in a box to be installed onto your computer's hard drive. Client/Server is a conglomeration of computer equipment, infrastructure, and software programs working together to accomplish computing tasks which enable their users to be more efficient and productive. Client/Server applications can be distinguished by the nature of the service or type of solutions they provide. Client/Server Computing is new technology that yields solutions to many data management problems faced by modern organizations.

Client/Server Computing: An Introduction, features objective evaluations and details of Client/Server development tools, used operating system, database management system and its mechanism in respect of Client/Server computing and network components used in order to build effective Client/Server applications.

Last but not the least, this work is primarily a joint work with a number of fellow teacher who have worked with us. My parents, wife Meera, and our children, Akanksha and Harsh. I am particularly grateful to Dr. A. P. Singh, Principal, Udai Pratap Inter College, Varanasi; Dr. D. S. Yadav, Sr. Lecturer, Department of Computer Science and Engineering, IET, Lucknow; Dr. A. K. Naiyak, Director IIBM, Patna, former President of IT and Computer Science Section of Indian Science Congress Association; Prof. A. K. Agrawal, Professor and Ex-Head of Department, Computer Science and Engineering IT, BHU, Varanasi and Mr. Manish Kumar Singh, Sr. Lecturer, Rajarshi School of Management and Technology for providing the necessary help to finish this work.

Suggestions and comments about the book are most welcome and can be sent by e-mail to scy@rediffmail.com.

Subhash Chandra Yadav

Contents

Preface *v*

1 INTRODUCTION **1–23**

1.1 What is Client/Server Computing? 1

1.1.1 A Server for Every Client 2

1.1.2 Client/Server: Fat or Thin 4

1.1.3 Client/Server: Stateless or Stateful 4

1.1.4 Servers and Mainframes 5

1.1.5 Client/Server Functions 7

1.1.6 Client/Server Topologies 7

1.1.7 Integration with Distributed Computing 8

1.1.8 Alternatives to Client/Server Systems 9

1.2 Classification of Client/Server Systems 9

1.2.1 Two-tier Client/Server Model 9

1.2.2 Three-tier Client/Server Model 12

1.2.2.1 Transaction Processing Monitors 15

1.2.2.2 Three-tier with Message Server 16

1.2.2.3 Three-tier with an Application Server 17

1.2.2.4 Three-tier with an ORB Architecture 17

1.2.2.5 Three-tier Architecture and Internet 17

1.2.3 N-tier Client/Server Model 18

1.3 Clients/Server— Advantages and Disadvantages 19

1.3.1 Advantages 19

1.3.2 Disadvantages 21

1.4 Misconceptions About Client/Server Computing 22

Exercise 1 23

2 DRIVING FORCES BEHIND CLIENT/SERVER COMPUTING 25–40

2.1 Introduction 25
2.2 Driving Forces 26
2.2.1 Business Perspective 26
2.2.2 Technology Perspective 28
2.3 Development of Client/Server Systems 29
2.3.1 Development Tools 30
2.3.2 Development Phases 30
2.4 Client/Server Standards 32
2.5 Client/Server Security 33
2.5.1 Emerging Client /Server Security Threats 33
2.5.2 Threats to Server 34
2.6 Organizational Expectations 34
2.7 Improving Performance of Client/Server Applications 36
2.8 Single System Image 37
2.9 Downsizing and Rightsizing 38
2.10 Client/Server Methodology 39
Exercise 2 40

3 ARCHITECTURES OF CLIENT/SERVER SYSTEMS 41–62

3.1 Introduction 41
3.2 Components 42
3.2.1 Interaction between the Components 43
3.2.2 Complex Client/Server Interactions 43
3.3 Principles behind Client/Server Systems 45
3.4 Client Components 46
3.5 Server Components 48
3.5.1 The Complexity of Servers 51
3.6 Communications Middleware Components 52
3.7 Architecture for Business Information System 55
3.7.1 Introduction 55
3.7.2 Three-Layer Architecture 56
3.7.3 General Forces 56
3.7.4 Distribution Pattern 58
3.8 Existing Client/Server Architecture 59
3.8.1 Mainframe-based Environment 59
3.8.2 LAN-based Environment 60
3.8.3 Internet-based Environment 60
Exercise 3 62

4 CLIENT/SERVER AND DATABASES 63–78

4.1 Introduction 63
4.2 Client/Server in Respect of Databases 64
4.2.1 Client/Server Databases 64
4.2.2 Client/Server Database Computing 65
4.3 Client/Server Database Architecture 66
4.4 Database Middleware Component 70
4.5 Access to Multiple Databases 71
4.6 Distributed Client/Server Database Systems 72
4.7 Distributed DBMS 74
4.8 Web/database System for Client/Server Applications 76
4.8.1 Web/database Vs Traditional Database 77
Exercise 4 78

5 CLIENT/SERVER APPLICATION COMPONENTS 79–104

5.1 Introduction 79
5.2 Technologies for Client/Server Application 79
5.3 Service of a Client/Server Application 80
5.4 Categories of Client/Server Applications 84
5.5 Client Services 85
5.5.1 Inter Process Communication 87
5.5.2 Remote Services 91
5.5.3 Window Services 92
5.5.4 Dynamic Data Exchange (DDE) 92
5.5.5 Object Linking and Embedding (OLE) 93
5.5.6 Common Object Request Broker Architecture (CORBA) 94
5.5.7 Print/Fax Services 95
5.5.8 Database Services 95
5.6 Server Services 96
5.7 Client/Server Application: Connectivity 100
5.7.1 Role and Mechanism of Middleware 101
5.8 Client/Server Application: Layered Architecture 102
5.8.1 Design Approach 102
5.8.2 Interface in Three Layers 103
Exercise 5 104

6 SYSTEM DEVELOPMENT 105–138

6.1 Hardware Requirements 105
6.1.1 PC Level Processing Units 105
6.1.2 Storage Devices 110
6.1.3 Network Protection Devices 115
6.1.4 Surge Protectors 117
6.1.5 RAID Technology 120
6.1.6 Server Specific Jargon 122
6.2 Software Requirements 124
6.2.1 Client OS 124
6.2.2 Server OS 124
6.2.3 Network OS 128
6.3 Communication Interface Technology 131
6.3.1 Network Interface Card 131
6.3.2 LAN Cabling 132
6.3.3 WAN 132
6.3.4 ATM 133
6.3.5 Ethernet 133
6.3.6 Token Ring 134
6.3.7 FDDI 135
6.3.8 TCP/IP 135
6.3.9 SNMP 135
6.3.10 NFS 136
6.3.11 SMTP 136
Exercise 6 137

7 TRAINING AND TESTING 139–156

7.1 Introduction 139
7.2 Technology Behind Training Delivery 140
7.2.1 Traditional Classroom 140
7.2.2 On-the-Job Training (OTJ) 141
7.2.3 Video Conferencing 141
7.2.4 Collaborative Tools 141
7.2.5 Virtual Groups and Event Calls 142
7.2.6 E-Learning 142
7.2.7 Web-based Training 142
7.2.8 Learning Management Systems (LMS) 143
7.2.9 Electronic Performance Support Systems (EPSS) 143

7.3 To Whom Training is Required? 143
7.3.1 System Administrator Training 143
7.3.2 DBA Training 144
7.3.3 Network Administrator Training 145
7.3.4 End-User and Technical Staff Training 146
7.3.5 GUI Applications Training 146
7.3.6 LAN/WAN Administration and Training Issues 148
7.4 Impact of Technology on Training 149
7.4.1 Client/Server Administration and Management 150
7.5 Client/Server Testing Technology 150
7.5.1 Client/Server Software 150
7.5.2 Client/Server Testing Techniques 151
7.5.3 Testing Aspects 152
7.5.4 Measures of Completeness 153
7.6 Testing Client/Server Application 153
Exercise 7 156

8 CLIENT/SERVER TECHNOLOGY AND WEB SERVICES 157–172

8.1 Introduction 157
8.2 What are Web Services? 158
8.2.1 Web Services History 158
8.2.2 Web Server Technology 158
8.2.3 Web Server 162
8.2.4 Web Server Communication 163
8.3 Role of Java for Client/Server on Web 164
8.4 Web Services and Client/Server/Browser – Server Technology 167
8.5 Client/Server Technology and Web Applications 168
8.6 Balanced Computing and the Server's Changing Role 171
Exercise 8 172

9 FUTURE OF THE CLIENT/SERVER COMPUTING 173–193

9.1 Introduction 173
9.2 Technology of the Next Generation 173
9.2.1 Networking 174
9.2.2 Development Tools 174
9.2.3 Processors and Servers 177
9.2.4 Paradigms 178

9.3 Enabling Technology 178
9.3.1 Expert Systems 178
9.3.2 Imaging 180
9.3.3 Point-of-Service 181
9.4 Client/Server Computing and the Intranet 181
9.4.1 Intranet 181
9.4.2 Is the Intranet Killing Client/Server? 182
9.4.3 Extranet 183
9.5 Future Perspectives 183
9.5.1 Job Security 183
9.5.2 Future Planning 184
9.5.3 Conclusion 184
9.6 Transformational System 185
9.6.1 Electronic Mail 185
9.6.2 Client/Server and User Security 186
9.6.3 Object-oriented Technology: CORBA 188
9.6.4 Electronic Data Interchange 192
Exercise 9 193

References **195–197**

Index **199–200**

1

Introduction

1.1 WHAT IS CLIENT/SERVER COMPUTING?

According to MIS terminology, Client/Server computing is new technology that yields solutions to many data management problems faced by modern organizations. The term Client/Server is used to describe a computing model for the development of computerized systems. This model is based on distribution of functions between two types of independent and autonomous processes: Server and Client. A Client is any process that requests specific services from the server process. A Server is a process that provides requested services for the Client. Client and Server processes can reside in same computer or in different computers linked by a network.

When Client and Server processes reside on two or more independent computers on a network, the Server can provide services for more than one Client. In addition, a client can request services from several servers on the network without regard to the location or the physical characteristics of the computer in which the Server process resides. The network ties the server and client together, providing the medium through which the clients and the server communicate. The Fig. 1.1 given below shows a basic Client/Server computing model.

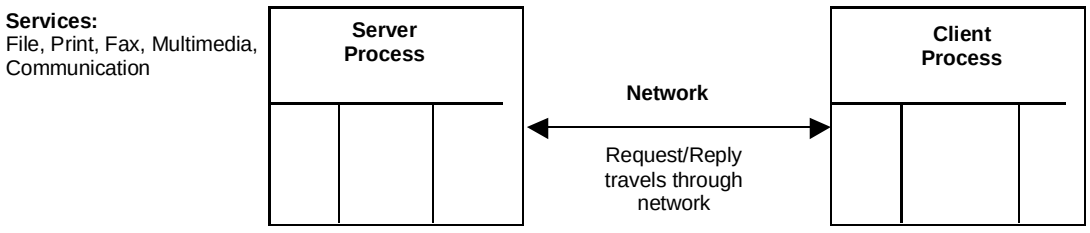


Fig.1.1: Basic Client/Server Computing Model

From the Fig. 1.1 it is clear that services can be provided by variety of computers in the network. The key point to Client/Server power is where the request processing takes place. For example: Client/Server Database. In case of Client/Server database system, the functionality is split between the server system and multiple clients such that networking of computers allows some tasks to be executed on the client system.

1.1.1 A Server for Every Client

A file server can store any type of data, and so on simpler systems, may be the only server necessary. On larger and more complicated systems, the server responsibility may be distributed among several different types of servers. In this section, we have discussed the purpose of various available server:

File Server

All the files reside on the server machine. File Server provides clients access to records within files from the server machine. File Servers are useful for sharing files across a network among the different client process requesting the services. The server process is somewhat primitive because of tends to demand many message exchanges over the network to find the requested data.

The examples of File servers are:

- UNIX: Network File Services (NFS) created by Sun Micro systems.
- Microsoft Windows "Map Drive" e.g., Rivier College's "P-drive".
- Samba: An open Source/Free Software suite that provides seamless file and print services to SMB/CIFS clients (i.e., Microsoft Windows clients).

Print Server

This machine manages user access to the shared output devices, such as printers. These are the earliest type of servers. Print services can run on a file server or on one or more separate print server machines.

Application Server

This machine manages access to centralized application software; for example, a shared database. When the user requests information from the database, the application server processes the request and returns the result of the process to the user.

Mail Server

This machine manages the flow of electronic mail, messaging, and communication with mainframe systems on large-scale networks.

Fax Server

Provides the facility to send and receive the Faxes through a single network connection. The Fax server can be a workstation with an installed FAX board and special software or a specialized device dedicated and designed for Fax Services. This machine manages flow of fax information to and from the network. It is similar to the mail server.

Directory Services Server

It is found on large-scale systems with data that is distributed throughout multiple servers. This machine functions as an organization manager, keeping track of what is stored where, enabling fast and reliable access to data in various locations.

Web Server

This machine stores and retrieves Internet (and intranet) data for the enterprise. Some documents, data, etc., reside on web servers. Web application provides access to documents and other data. "Thin" clients typically use a web browser to request those documents. Such servers shares documents across intranets, or across the Internet (or extranets). The most commonly used protocol is HTTP (Hyper Text Transfer Protocol). Web application servers are now augmenting simple web servers. The examples of web application servers are Microsoft's Internet Information Server (IIS), Netscape's iPlanet IBM's WebSphere, BEA's WebLogic and Oracle Application Server.

Database Server

Data resides on server, in the form of a SQL database. Database server provides access to data to clients, in response to SQL requests. It shares the data residing in a database across a network. Database Server has more efficient protocol than File Server. The Database Server receives SQL requests and processes them and returning only the requested data; therefore the client doesn't have to deal with irrelevant data. However, the client does have to implement SQL application code. The example of database server is: Oracle9i database server.

Transaction Servers

The data and remote procedures reside on the server. The Server provides access to high-level functions, and implements efficient transaction processing. It shares data and high-level functions across a network. Transaction servers are often used to implement Online Transaction Processing (OLTP) in high-performance applications. A transaction server utilizes a more efficient protocol in comparison to a Database Server. The transaction Server receives high-level function request from the clients and it implements that function. Often it needs to return less information to the client than a Database Server. Examples of the Transaction servers mainly categorized as

- TP-Light with Database Stored Procedures like Oracle, Microsoft SQL Server etc.
- TP-Heavy with TP Monitors like BEA Tuxedo, IBM CICS/TX Series.

Groupware Servers

Liable to store semi-structured information like text, image, mail, bulletin boards, flow of work. Groupware Server provides services, which put people in contact with other people, that is because "groupware" is an ill-defined classification protocol differing from product to product. For Example: Lotus Notes/Domino and Microsoft Exchange.

Object Application Servers

Communicating distributed objects reside on the server. The object server primarily provides access to those objects from the designated client objects. The object Application Servers are responsible for sharing distributed objects across the network. Object Application Servers use the protocols that are usually some kind of Object Request Broker (ORB). Each distributed object can have one or more remote methods. ORB locates an instance of the object server class, invokes the requested method, and returns the results to the client object. Object Application Server provides an ORB and application servers to implement this. For example:

- Common Object Request Broker Architecture (CORBA): Iona's Orbix, Borland's Visibroker.
- Microsoft's Distributed Component Object Model (DCOM), aka COM+.
- Microsoft Transaction Server (MTS).

1.1.2 Client/Server: Fat or Thin

A Client or a Server is so named depending on the extent to which the processing is shared between the client and server. A thin client is one that conducts a minimum of processing on the client side while a fat client is one that carries a relatively larger proportion of processing load. The concept of Fat Clients or Fat Servers is given by one of the important criterion, that is, how much of an application is placed at the client end vs. the server end.

Fat Clients: This architecture places more application functionality *on the client machine(s).* They are used in traditional of Client/Server models. Their use can be a maintenance headache for Client/Server systems.

Fat Servers: This architecture places more application functionality *on the server machine(s).* Typically, the server provides more abstract, higher level services. The current trend is more towards fat servers in Client/Server Systems. In that case, the client is often found using a fast web browser. The biggest advantage of using the fat server is that it is easier to manage because only the software on the servers needs to be changed, whereas updating potentially thousands of client machines is a real headache.

1.1.3 Client/Server: Stateless or Stateful

A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request. The biggest advantage of stateless is that it simplifies the server design because it does not need to dynamically allocate storage to deal with conversations in progress or worry about freeing it if a client dies in mid-transaction. There is also one disadvantage that it may be necessary to include more information in each request and this extra information will need to be interpreted by the server each time. An example of a stateless server is a World Wide Web server. With the exception of cookies, these take in requests (URLs) which completely specify the required document and do not

require any context or memory of previous requests contrast this with a traditional FTP server which conducts an interactive session with the user. A request to the server for a file can assume that the user has been authenticated and that the current directory and file transfer mode have been set. The Gopher protocol and Gopher + are both designed to be stateless.

Stateful Server

Client data (state) information are maintained by server on status of ongoing interaction with clients and the server remembers what client requested previously and at last maintains the information as an incremental reply for each request.

The advantages of stateful server is that requests are more efficiently handled and are of smaller in size. Some disadvantages are their like state information becomes invalid when messages are unreliable. Another disadvantage is that if clients crash (or reboot) frequently, state information may exhaust server's memory. The best example of stateful server is remote file server.

Stateless vs Stateful Servers

There are some comparative analysis about stateless and stateful servers.

* A stateful server remembers client data (state) from one request to the next.
* A stateless server keeps no state information. Using a stateless file server, the client must specify complete file names in each request specify location for reading or writing and re-authenticate for each request.
* Using a stateful file server, the client can send less data with each request. A stateful server is simpler.

On the other hand, a stateless server is more robust and lost connections can't leave a file in an invalid state rebooting the server does not lose state information rebooting the client does not confuse a stateless server.

1.1.4 Servers and Mainframes

From a hardware perspective, a mainframe is not greatly different from a personal computer. The CPU inside a mainframe was, however, much faster than a personal computer. In fact, what a mainframe most closely resembled was a LAN. A mainframe was 'larger' in terms of:

* The raw speed expressed in instructions per second, or cycles.
* The amount of memory that could be addressed directly by a program.

Mainframes are the monstrous computer system that deals mainly the business functions and technically these giant machines will run MVS, IMS and VSAM operating systems. There is a common believe that a mainframe is 'database'. There are many reasons behind this belief:

* Many servers are either file or database servers running sophisticated database such as Sybase, Oracle and DB2.

* These servers connect to the mainframe primarily to access databases.
* Organisations use servers specifically to replace mainframe databases.
* Organisations keep applications on the mainframe usually for better database performance, integrity and functionality.

Mainframe users argue that in the long run, a mainframe is at least as good a server as a PC, and perhaps even better. And because the mainframe portrayed as a better server than a PC, the picture is clear: PC servers and mainframe servers compete at the back-end both are essentially databases.

There is some controversy as to whether servers will eventually replace mainframes. They may, but not in the near future. Mainframes still serve the purpose in managing the complex business rules of very large organizations and enterprises that are spread out over a very large area. But the increasing processing power of servers combined with their lower costs makes them the logical replacement to mainframe-based systems in the future.

In the meanwhile, Client/Server networks will often find it necessary to connect to mainframe-based systems. This is because some data can only be found in the mainframe environment, usually because the business rules for handling it are sufficiently complex or because the data itself is massive or sensitive enough that as a practical matter it remains stored there.

Connection to a mainframe requires some form of network – like access. Even if you are using a telephone and modem as your access hardware, you still require special software to make your workstation appear to the mainframe to be just another network terminal. Many vendors can provide the necessary software to handle this type of network extension.

A very natural question at this stage is: How do Client/Server Systems differ from Mainframe Systems?

The extent of the separation of data processing task is the key difference.

In mainframe systems all the processing takes place on the mainframe and usually dumb terminals are used to display the data screens. These terminals do not have autonomy.

On the other hand, the Client/Server environment provides a clear separation of server and client processes, both processes being autonomous. The relationship between client and server is many to many.

Various other factors, which can have, prime considerations to differentiate the mainframe and Client/Server systems:

- **Application development:** Mainframe systems are over structured, time-consuming and create application backlogs. On the other hand, PC-based Client/Server systems are flexible, have rapid application development and have better productivity tools.
- **Data manipulation:** Mainframe systems have very limited data manipulation capabilities whereas these techniques are very flexible in the case of Client/Server systems.

- **System management:** Mainframe systems are known to be integrated systems but in the case of Client/Server systems only few tools are available for system management.
- **Security:** Mainframe systems are highly centralized whether as Client/Server systems are relaxed or decentralized.
- **End user platform:** Mainframe systems comprise of dumb terminals, are character-based, single task oriented and of limited productivity. On the other hand, Client/Server systems are intelligent PC's with graphical user interface having multitasking OS with better productivity tools.

1.1.5 Client/Server Functions

The main operations of the client system are listed below:

- Managing the user interface.
- Accepts and checks the syntax of user inputs.
- Processes application logic.
- Generates database request and transmits to server.
- Passes response back to server.

The main operations of the server are listed below:

- Accepts and processes database requests from client.
- Checks authorization.
- Ensures that integrity constraints are not violated.
- Performs query/update processing and transmits responses to client.
- Maintains system catalogue.
- Provide concurrent database access.
- Provides recovery control.

1.1.6 Client/Server Topologies

A Client/Server topology refers to the physical layout of the Client/Server network in which all the clients and servers are connected to each other. This includes all the workstations (clients) and the servers. The possible Client/Server topological design and strategies used are as follows:

(*i*) Single client, single server

(*ii*) Multiple clients, single server

(*iii*) Multiple clients, multiple servers

(*i*) **Single client, single server:** This topology is shown in the Fig. 1.2 given below. In this topology, one client is directly connected to one server.

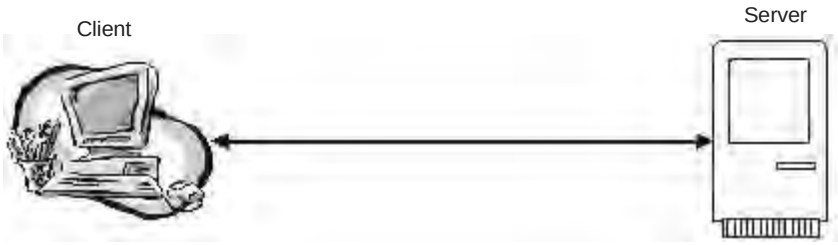


Fig.1.2: Single Client, Single Server

(*ii*) **Multiple clients, single server:** This topology is shown in the Fig. 1.3 given below. In this topology, several clients are directly connected to only one server.

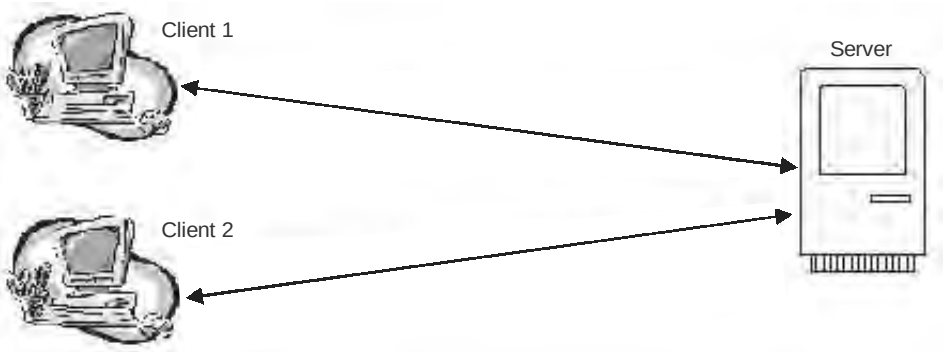


Fig.1.3: Multiple Clients, Single Server

(*iii*) **Multiple clients, multiple servers:** This topology is shown in the following Fig. 1.4 In this topology several clients are connected to several servers.

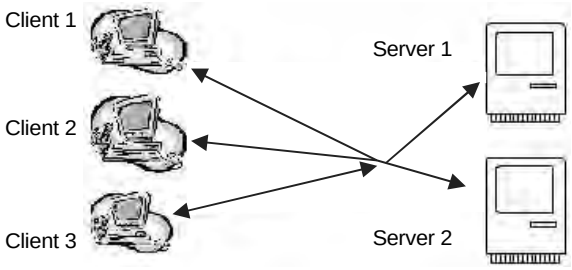


Fig.1.4: Multiple Clients, Multiple Servers

1.1.7 Integration with Distributed Computing

Distributed computing is the term used for implementation of technologies across heterogeneous environments. For operating systems, heterogeneous computing means the ability to communicate with other operating systems and protocols. Distributed computing is a complex architecture. It involves rearchitecture of applications, redevelopment of systems and increased efficiency in maintaining a network as a whole. Many distributed

nodes work on behalf of one requesting client. This makes the system fault tolerant and decentralized, which is an obvious advantage over centralized systems. For the technology to become effective and revolutionary, developers of distributed applications have to do everything possible to minimize the complexity of development and maintenance and integrate their software with disparate platforms. Client/Server application designing necessitates the modularization of applications and their functions into discrete components. These components must be bounded only by encapsulated data and functions that may be moved between the systems. This design model gives Client/Server software more adaptability and flexibility.

1.1.8 Alternatives to Client/Server Systems

There are various client/server projects are running in industry by various companies. Before committing a project to Client/Server, some alternatives can be considered that includes:

- Movement of an existing mainframe application to a smaller hardware platforms, for examples IBM's ICCS transaction processing to an AS/400 or an OS/2 LAN Server.
- Replacement of mainframes computer terminals with PCs that is able to emulate terminals.
- Replacing an existing mainframe system with a packaged system that does the job better.
- Beautifying an existing mainframe application by adding a GUI front-end to it. There are programs available specifically to do this.

1.2 CLASSIFICATION OF CLIENT/SERVER SYSTEMS

Broadly, there are three types of Client/Server systems in existence.

(*i*) Two-tier

(*ii*) Three-tier

(*iii*) N-Tier

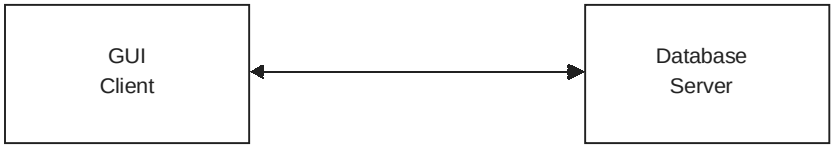
1.2.1 Two-tier Client/Server Model

The application processing is done separately for database queries and updates and for business logic processing and user interface presentation. Usually, the network binds the back-end of an application to the front-end, although both tiers can be present on the same hardware.

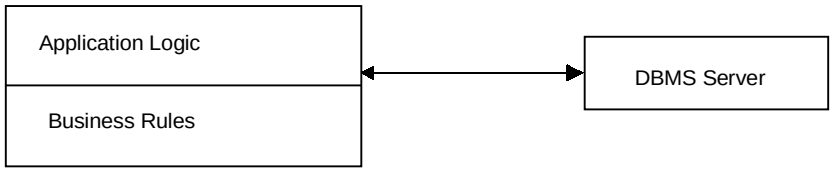
Sometimes, the application logic (the real business logic) is located in both the client program and in the database itself. Quiet often, the business logic is merged into the presentation logic on the client side. As a result, code maintenance and reusability become difficult to achieve on the client side. On the database side, logic is often developed using stored procedures.

In the two-tier architecture, if the Client/Server application has a number of business rules needed to be processed, then those rules can reside at either the Client or at the Server. The Fig. 1.5 below clarifies this situation.

(*a*) Centralized Two-tier Representation:



(*b*) Business Rules Residing on the Client:



(*c*) Business Rules Residing on the Server:

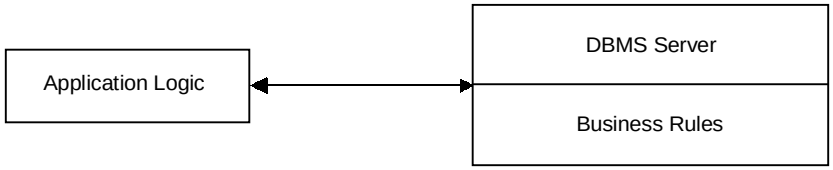


Fig.1.5: The Two-tier Approach Illustrated

The architecture of any client/server environment is by definition at least a two-tier system, the client being the first tier and the server being the second.

The Client requests services directly from server i.e. client communicates directly with the server without the help of another server or server process. The Fig. 1.6 (at the end of this section) illustrates a two-tier Client/Server model.

In a typical two-tier implementation, SQL statements are issued by the application and then handed on by the driver to the database for execution. The results are then sent back via the same mechanism, but in the reverse direction. It is the responsibility of the driver (ODBC) to present the SQL statement to the database in a form that the database understands.

There are several advantages of two-tier systems:

- Availability of well-integrated PC-based tools like, Power Builder, MS Access, 4 GL tools provided by the RDBMS manufacturer, remote SQL, ODBC.
- Tools are relatively inexpensive.
- Least complicated to implement.
- PC-based tools show Rapid Application Development (RAD) i.e., the application can be developed in a comparatively short time.
- The 2-tier Client/Server provides much more attractive graphical user interface (GUI) applications than was possible with earlier technology.

- Architecture maintains a persistent connection between the client and database, thereby eliminating overhead associated with the opening and closing of connections.
- Faster than three-tier implementation.
- Offers a great deal of flexibility and simplicity in management.

Conversely, a two-tier architecture has some disadvantages:

- As the application development is done on client side, maintenance cost of application, as well as client side tools etc. is expensive. That is why in 2-tier architecture the client is called 'fat client'.
- Increased network load: Since actual processing of data takes on the remote client, the data has to be transported over the network. This leads to the increased network stress.
- Applications are loaded on individual PC i.e. each application is bound to an individual PC. For this reason, the application logic cannot be reused.
- Due to dynamic business scenario, business processes/logic have to be changed. These changed processes have to be implemented in all individual PCs. Not only that, the programs have to undergo quality control to check whether all the programs generate the same result or not.
- Software distribution procedure is complicated in 2-tier Client/Server model. As all the application logic is executed on the PCs, all these machine have to be updated in case of a new release. The procedure is complicated, expensive, prone to errors and time consuming.
- PCs are considered to be weak in terms of security i.e., they are relatively easy to crack.
- Most currently available drivers require that native libraries be loaded on a client machine.
- Load configurations must be maintained for native code if required by the driver.
- Problem areas are encountered upon implementing this architecture on the Internet.

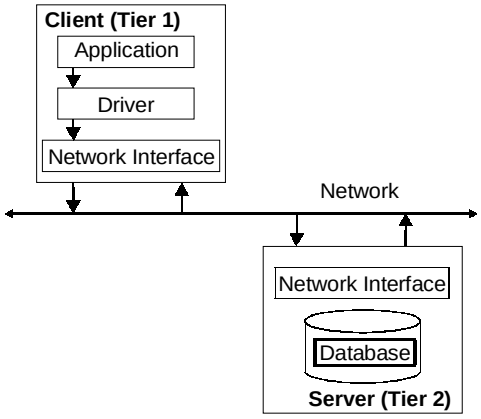


Fig. 1.6: Two-tier Client/Server Model

1.2.2 Three-tier Client/Server Model

Reusability is hard to achieve if pieces of business logic must be distributed across systems and several databases are involved. To avoid embedding the application's logic at both the database side and the client side, a third software tier is inserted in between. In the three-tier architecture, most of the business logic is located in the middle tier (here business logic is encapsulated as a component in a separate tier). In this structure, when the business activity or business rules change, only the middle tier must be modified.

In three-tier architecture application responsibilities are divided into three logical categories (in other words, the business system must provide three types of main services).

- **Presentation (GUI) or user services:** Include maintaining the graphical user interface and generating what users see on the monitor. Presentation Logic dealing with:
  - Screen formatting
  - Windows management
  - Input editing
  - What-if analysis
- **Application services or business rules:** These include executing applications and controlling program flow. Business logic dealing with:
  - Domain and range validation
  - Data dependency validation
  - Request/response architecture of Inter Process Communication level
- **Database services or data server:** Which refers to the management of underlying databases. Server logic deals with:
  - Data access
  - Data management
  - Data security
  - SQL parsing

Based on these three components, the three-tier architecture of Client/Server system is shown in fig. 1.8 below. In three-tier model, a third server is employed to handle requests from the client and then pass them off to the database server. The third server acts as proxy for all client requests. Or, in other words we can say:

"In three-tier client/server system the client request are handled by intermediate servers which coordinate the execution of the client request with subordinate servers."

All client requests for the database are routed through the proxy server, thereby creating a more secure environment for your database.

In two-tier environment, we can say that the client uses a driver to translate the client's request into a database native library call. In a three-tier environment, the driver translates the request into a "network" protocol and then makes a request via the proxy server. Figure 1.8 represents the three-tier Client/Server model.

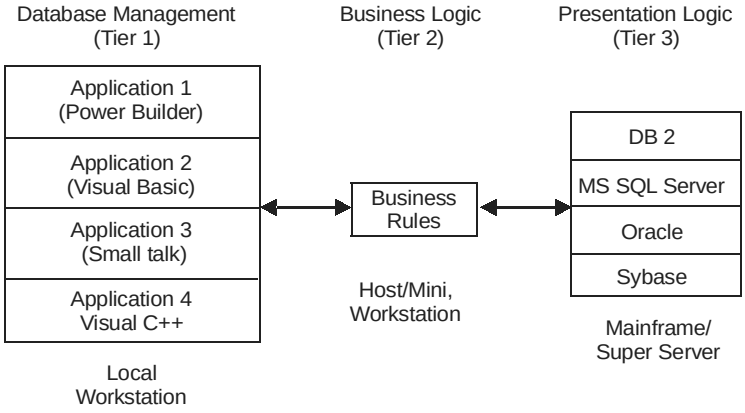


Fig.1.7: Three-tier Architecture

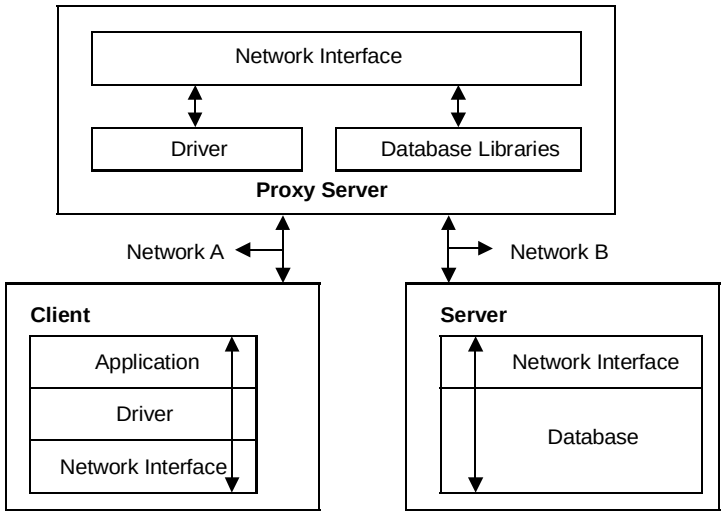


Fig.1.8: Three-tier Client/Server Model

The proxy server makes the database request on behalf of the client and passes the results back after they have been serviced by the database. This approach eliminates the need for DBMS to be located on the same server. There are a couple of drawbacks to this model. One is that it requires that a small server process (listener) be set up on the middle server. Secondly, it requires all your client requests be transmitted into a "network" protocol.

First-tier (client-tier): The main responsibility of this tier is to receive user events and to control the user interface and presentation of data. As most of the software is removed from the client, the client is called "Thin Client". Mainly browser and presentation code resides on this tier.

Second-tier (application-server-tier): The complex application logic is loaded here and available to the client tier on request from client. This level forms the central key

towards solving the 2-tier problem. This tier can protect direct access of data. Object oriented analysis aims in this tier to record and abstract business processing in business projects. This way it is possible to map this tier directly from the case tools that support object oriented analysis.

Three-tier (database-server-tier): This tier is responsible for data storage. This server mostly operates on a relational database.

The boundaries between tiers are logical. One can run 3-tiers in one and the same machine. The important fact is that the system is neatly structured and well-planned definitions of the software boundaries exist between the different tiers. Some of the advantages of using three-tier model include:

- Application maintenance is centralized with the transfer of the business logic for many end users into a single application server. This eliminates the concern of software distribution that are problematic in the traditional two-tier Client/Server model.
- Clear separation of user-interface-control and data presentation from application-logic. Through this separation more clients are able to have access to a wide variety of server applications. The two main advantages for client-applications are clear: quicker development through the reuse of pre-built business-logic components and a shorter test phase, because the server-components have already been tested.
- Many users are able to access a wide variety of server applications, as all application logic are loaded in the applications server.
- As a rule servers are "trusted" systems. Their authorization is simpler than that of thousands of "untrusted" client-PCs. Data protection and security is simpler to obtain. Therefore, it makes sense to run critical business processes that work with security sensitive data, on the server.
- Redefinition of the storage strategy won't influence the clients. RDBMS' offer a certain independence from storage details for the clients. However, cases like changing table attributes make it necessary to adapt the client's application. In the future, even radical changes, like switching from an RDBMS to an OODBMS, won't influence the client. In well-designed systems, the client still accesses data over a stable and well-designed interface, which encapsulates all the storage details.
- Load balancing is easier with the separation of the core business logic from the database server.
- Dynamic load balancing: if bottlenecks in terms of performance occur, the server process can be moved to other servers at runtime.
- Business objects and data storage should be brought as close together as possible. Ideally, they should be together physically on the same server. This way network load for complex access can be reduced.
- The need for less expensive hardware because the client is 'thin'.
- Change management is easier and faster to execute. This is because a component/ program logic/business logic is implemented on the server rather than furnishing numerous PCs with new program versions.

- The added modularity makes it easier to modify or replace one tier without affecting the other tier.
- Clients do not need to have native libraries loaded locally.
- Drivers can be managed centrally.
- Your database server does not have to be directly visible to the Internet.

An additional advantage is that the three-tier architecture maps quite naturally to the Web environment, with a Web browser acting as the 'thin' client, and a Web server acting as the application server. The three-tier architecture can be easily extended to N-tier, with additional tiers added to provide more flexibility and scalability. For example, the middle tier of the three-tier architecture could be split into two, with one tier for the Web server and another for the application server. Some disadvantages are:

- The client does not maintain a persistent database connection.
- A separate proxy server may be required.
- The network protocol used by the driver may be proprietary.
- You may see increased network traffic if a separate proxy server is used.

1.2.2.1 Transaction Processing Monitors

It is the extension of the two-tier Client/Server architecture that splits the functionality of the 'fat' client into two. In the three-tier Client/Server architecture, the 'thin' client handles the user interface only whereas the middle layer handles the application logic. The third layer is still a database server. This three-tier architecture has proved popular in more environments, such as the Internet and company Intranets where a Web browser can be used as a client. It is also an important architecture for TPM.

A Transaction Processing Monitor is a program that controls data transfer between client and server in order to provide consistent environment, particularly for online transaction processing (OLTP).

Complex applications are often built on top of several resource managers (such as DBMS, operating system, user interface, and messaging software). A Transaction Processing Monitor or TP Monitor is a middleware component that provides access to the services of a number of resource managers and provides a uniform interface for programmers who are developing transactional software. Figure 1.9 illustrates how a TP Monitor forms the middle tier of three-tier architecture. The advantages associated with TP Monitors are as given below:

Transaction Routing: TP monitor can increase scalability by directing transactions to specific DBMS's.

Managing Distributed Transaction: The TP Monitor can manage transactions that require access to data held in multiple, possibly heterogeneous, DBMSs. For example, a transaction may require to update data item held in an oracle DBMS at site 1, an Informix DBMS at site 2, and an IMS DBMS at site 3. TP Monitors normally control transactions using the X/Open Distributed transaction processing (DTP) standards. A DBMS that

support this standard can function as a resource manager under the control of a TP Monitor acting as a transaction manager.

Load balancing: The TP Monitor can balance client requests across multiple DBMS's on one or more computers by directing client services calls to the least loaded server. In addition, it can dynamically bring in additional DBMSs as required to provide the necessary performance.

Funneling: In an environment with a large number of users, it may sometimes be difficult for all users to be logged on simultaneously to the DBMS. Instead of each user connecting to the DBMS, the TP Monitor can establish connections with DBMS's as and when required, and can funnel user requests through these connections. This allows a large number of users to access the available DBMSs with a potentially smaller number of connections, which in turn would mean less resource usages.

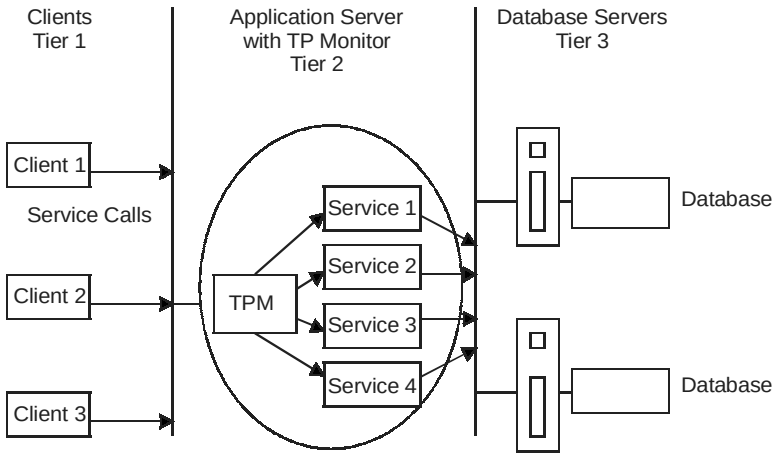


Fig.1.9: Middleware Component of TPM

Increased reliability: The TP Monitor acts as transaction manager, performing the necessary action to maintain the consistency of database, with the DBMS acting as a resource manager. If the DBMS fails, the TP Monitor may be able to resubmit the transaction to another DBMS or can hold the transaction until the DBMS becomes available again.

A TP Monitor is typically used in environments with a very heavy volume of transaction, where the TP Monitor can be used to offload processes from the DBMS server. Prominent examples of TP Monitors include CICS and Encina from IBM (which are primarily used on IBM AIX or Windows NT and bundled now in the IBM TXSeries) and Tuxido from BEA system.

1.2.2.2 Three-tier with Message Server

Messaging is another way to implement the three-tier architecture. Messages are prioritized and processed asynchronously. Messages consist of headers that contain priority information, and the address and identification number. The message server connects to

the relational DBMS and other data sources. The difference between the TP monitor technology and the message server is that the message server architecture focuses on intelligent messages, whereas the TP Monitor environment has the intelligence in the monitor, and treats transactions as dumb data packets. Messaging systems are good solutions for wireless infrastructures.

1.2.2.3 Three-tier with an Application Server

The three-tier application server architecture allocates the main body of an application to run on a shared host rather than in the user system interface client environment. The application server does not drive the GUIs; rather it shares business logic, computations, and a data retrieval engine. Advantages are that with less software on the client there is less security to worry about, applications are more scalable, and support and installation costs are less on a single server than maintaining each on a desktop client. The application server design should be used when security, scalability, and cost are major considerations.

1.2.2.4 Three-tier with an ORB Architecture

Currently, work is going on in the industry towards developing standards to improve interoperability and determine what the common Object Request Broker (ORB) will be. Developing client/server systems using technologies that support distributed objects holds great promise, as these technologies support interoperability across languages and platforms, as well as enhancing maintainability and adaptability of the system. There are two-prominent distributed object technologies at present:

- Common Object Request Broker Architecture (CORBA).
- COM/DCOM (Component Object Model/Distributed Component Object Model).

Standards are being developed to improve interoperability between CORBA and COM/DCOM. The Object Management Group (OMG) has developed a mapping between CORBA and COM/DCOM that is supported by several products.

Distributed/collaborative enterprise architecture: The distributed/collaborative enterprise architecture emerged in 1993. This software architecture is based on Object Request Broker (ORB) technology, but goes further than the Common Object Request Broker Architecture (CORBA) by using shared, reusable business models (not just objects) on an enterprise-wide scale. The benefit of this architectural approach is that standardized business object models and distributed object computing are combined to give an organization flexibility to improve effectiveness organizationally, operationally, and technologically. An enterprise is defined here as a system comprised of multiple business systems or subsystems. Distributed/collaborative enterprise architectures are limited by a lack of commercially-available object orientation analysis and design method tools that focus on applications.

1.2.2.5 Three-tier Architecture and Internet

With the rapid development of Internet and web technology, Client/Server applications running over Internets and Intranets are becoming a new type of distributed computing. A typical web application uses the following 3-tier architecture.

- The user interface runs on the desktop as client.
- The client is connected (through one or more immediate server links) to a web server, which may be a storehouse for downloadable applets (Software components).
- This web server is, in turn, is supported by a database server which keeps track of information specific to the client interest and history.

These web applications rely on Internet standards (HTTP, HTML, XML etc.) as well as distributed objects programming languages.

1.2.3 N-tier Client/Server Model

N-tier computing obliges developer to design components according to a business schema that represents entities, relationship, activities roles, and rules, thereby enabling them to distribute functionality across logical and physical tiers, allowing better utilization of hardware and platform resources, as well as sharing of those resources and the components that they support to serve several large applications at the same time.

Another aspect of splitting tiers is that application developers and administrators are able to identify bottlenecks and throw hardware at them to enable load-balancing and fail-over of certain nodes. The splitting may be between application logic components, security logic, and presentation logic, computational-intensive and I/O-intensive components and so on. The most common approach used when designing N-tier system is the three-tier architecture. Three-tier and N-tier notations are similar, although N-tier architecture provides finer-grained layers. Architectures often decide to layout much more than three -tiers to deploy services (An infrastructure that supports three-tier is often made of several machines and services whose functionalities aren't part of the three-tier design).

Figure 1.10 shown below depicts the N-tier architecture.

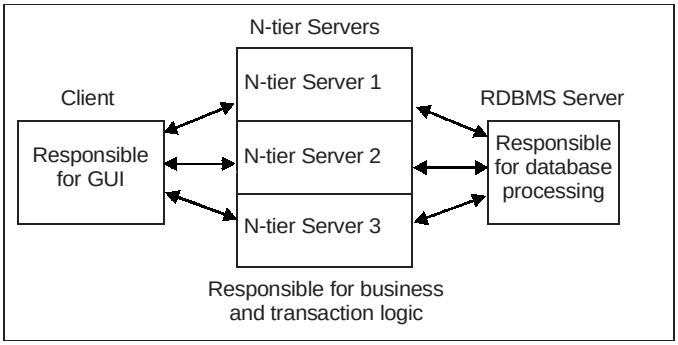


Fig.1.10: N-tier Architecture

N-tier computing provides many advantages over traditional two-tier or single-tier design, which includes the following:

- Overall performance has been improved.
- The business logic is centralized.
- Enhanced security level is attained.

An alternative to N-tier computing includes *fat server/fat client.* A *fat server* locates business logic within the RDBMS on the server. The client issues remote procedure calls to the server to execute the process. Fat servers are the best suited for structured and consistent business logic, such as online transaction processing (OLTP). Modern RDBMS products support fat servers through stored procedures, column rules, triggers, and other methods.

A *fat client* embeds business logic in the application at the client level. Although a fat client is more flexible than a fat server, it increases network traffic. The fat client approach is used when business logic is loosely structured or when it is too complicated to implement at the middle-tier level. Additionally, fat client development tools, such as 4GL languages, sometimes offer more robust programming features than do middle-tier programming tools. Decision support and ad-hoc systems are often fat client based.

1.3 CLIENTS/SERVER—ADVANTAGES AND DISADVANTAGES

1.3.1 Advantages

There are various advantages associated with Client/Server computing model.

(*i*) **Performance and reduced workload:** Processing is distributed among the client and server unlike the traditional PC database, the speed of DBMS is not tied to the speed of the workstation as the bulk of the database processing is done at the back-end. The workstation only has to be capable of running the front-end software, which extends the usable lifetime of older PC's. This also has the effect of reducing the load on the network that connects the workstation; instead of sending the entire database file back and forth on the wire, the network traffic is reduced to queries to and responses from the database server. Some database servers can even store and run procedures and queries on the server itself, reducing the traffic even more.

(*ii*) **Workstation independence:** Users are not limited to one type of system or platform. In an ORACLE-based Client/Server system the workstations can be IBM – compatible PCs, Macintoshes, UNIX workstations, or any combinations of the three. In addition, they can run any of a number of operating systems such as MS-DOS, Windows, IBM's OS/2, Apple's System 7 etc. That is, application independence is achieved as the workstations don't all need to use the same DBMS application software. Users can continue to use familiar software to access the database, and developers can design front-ends tailored to the workstation on which the software will run, or to the needs of the users running them.

(*iii*) **System interoperability:** Client/Server computing not only allows one component to be changed, it also makes it is possible for different type of components systems (client, network or server) to work together.

(*iv*) Scalability: The modular nature of the Client/Server system may be replaced without adversely affecting the rest of the system. For example, it is possible to upgrade the server to a more powerful machine with no visible changes to the end user. This ability to change component system makes Client/Server systems especially receptive to new technologies in both hardware and software.

(*v*) Data integrity: Client/Server system preserves the data integrity, DBMS can provide number of services that protect data like, encrypted file storage, real time backup (while the database is being accessed), disk mirroring (where the data is automatically written to duplicate database on another partition of same hard disk drive), disk duplexing (where the data is automatically written to a duplicate database on a different hard disk drive), transaction processing that keeps the track changes made to the database and corrects problems in case the server crashes. (Transaction processing is a method by which the DBMS keeps a running log of all the modifications made to the database over a period of time).

(*vi*) Data accessibility (enhanced data sharing): Since the server component holds most of data in a centralized location, multiple users can access and work on the data simultaneously.

(*vii*) System administration (centralized management): Client/Server environment is very manageable. Since data is centralized, data management can be centralized. Some of the system administration functions are security, data integrity and back up recovery.

(*viii*) Integrated services: In Client/Server model all information that the client is entitled to use is available at the desktop, through desktop interface, there is no need to change into a terminal mode or to logon into another processor to access information. The desktop tools – e-mail, spread sheet, presentation graphics, and word processing are available and can be used to deal with the information provided by application and database server's resident on the network. Desktop user can use their desktop tools in conjunction with information made available from the corporate systems to produce new and useful information using the facilities DDE/OLE, Object-oriented design.

(*ix*) Sharing resources among diverse platforms: Client/Server model provides opportunities to achieve open system computing. Applications can be created and implemented without much conversance with hardware and software. Thus, users may obtain client services and transparent access to the services provided by database, communications, and application servers. There are two ways for Client/Server application operation:

- They can provide data entry, storage, and reporting by using a distributed set of clients and servers.
- The existence of a mainframe host is totally masked from the workstation developer by the use of standard interface such as SQL.

(*x*) Masked physical data access: SQL is used for data access from database stored anywhere in the network, from the local PC, local server or WAN server, support with the developer and user using the same data request. The only noticeable difference may be performance degradation if the network bandwidth is inadequate. Data may be accessed from CD-ROM, HDD, Magnetic disk, and optical disk with same SQL statements. Logical tables can be accessed without any knowledge of the ordering of column. Several tables may be joined to create a new logical table for application program manipulation without regard to its physical storage format.

(*xi*) Location independence of data processing: Users log into an application from the desktop with no concern for the location or technology of the processors involved. In the current user centered word, the desktop provides the point of access to the workgroup and enterprise services without regard to the platform of application execution. Standard services such as login, security, navigation, help, and error recovery are provided consistently amongst all applications. Developers today are provided with considerable independence. Data is accessed through SQL without regard to the hardware or OS location providing the data. The developer of business logic deals with a standard process logic syntax without considering the physical platform.

(*xii*) Reduced operating cost: Computer hardware and software costs are on a continually downward spiral, which means that computing value is ever increasing. Client/Server computing offers a way to cash in on this bonanza by replacing expensive large systems with less expensive smaller ones networked together.

(*xiii*) Reduced hardware cost: Hardware costs may be reduced, as it is only the server that requires storage and processing power sufficient to store and manage the application.

(*xiv*) Communication costs are reduced: Applications carry out part of the operations on the client and send only request for database access across the network, resulting in less data being sent across the network.

1.3.2 Disadvantages

There are various disadvantages associated with the Client/Server computing model.

(*i*) Maintenance cost: Major disadvantages of Client/Server computing is the increased cost of administrative and support personnel to maintain the database server. In the case of a small network, the network administrator can usually handle the duties of maintaining the database server, controlling the user access to it, and supporting the front-end applications. However, the number of database server users rises, or as the database itself grows in size, it usually becomes necessary to hire a database administrator just to run the DBMS and support the front-ends.

(*ii*) Training cost: Training can also add to the start-up costs as the DBMS may run on an operating system that the support personnel are unfamiliar with.

(*iii*) **Hardware cost:** There is also an increase in hardware costs. While many of the Client/Server database run under the common operating systems (Netware, OS/2 and Unix) and most of the vendors claim that the DBMS can run on the same hardware side by side with the file server software. It usually makes sense from the performance and data integrity aspects to have the database server running on its own dedicated machine. This usually means purchasing a high-powered platform with a large amount of RAM and hard disk space.

(*iv*) **Software cost:** The overall cost of the software is usually higher than that of traditional PC based multi-user DBMS.

(*v*) **Complexity:** With so many different parts comprising the entire Client/Server, i.e., the more are the pieces, which comprise the system the more things that can go wrong or fail. It is also harder to pinpoint problems when the worst does occur and the system crashes. It can take longer to get everything set up and working in the first place. This is compounded by the general lack of experience and expertise of potential support personnel and programmers, due to the relative newness of the technology.

Making a change to the structure of database also has a ripple effect throughout the different front-ends. It becomes a longer and more complex process to make the necessary changes to the different front-end applications, and it is also harder to keep all of them in synchronization without seriously disrupting the user's access to the database.

1.4 MISCONCEPTIONS ABOUT CLIENT/SERVER COMPUTING

Client/Server technology can be stated to be an "architecture in which a system's functionality and its processing are divided between the client PC (Front-end) and database server (back-end)." This statement restricts the functionality of Client/Server software to mere retrieval and maintenance of data and creates many misconceptions regarding this technology, such as:

(*i*) Graphical user interface is supposed to be a necessity for presentation of application logic. As in the case of X-Windows graphical user interface, the implementation comprises both client and server components that may run on the same and different physical computers. An X-Windows uses Client/Server as architecture. This however, does not imply that Client/Server must use GUI. Client/Server logic remains independent of its presentation to the user.

(*ii*) Client/Server software is not always database centric. Client/Server computing does not require a database, although in today's computing environment Client/Server is synonymous with databases. RDBMS packages are the most popular Client/Server applications. A major disadvantage of Client/Server technology is that it can be data centric and the developers can exploit these capabilities.

(*iii*) Client/Server technology does not provide code reuse. Tools that help create Client/Server applications may provide this benefit. Client/Server application build on component-based modeling enhanced code reusability.

(*iv*) Client/Server designing is not event-driven. Client/Server technology merges very well with event-driven systems but the latter are not a requirement for Client/Server. Client/Server application designing involves architecture of software that has innate features of portability with respect to communication pattern, and a well-build non-monolithic code with division of functionality and processing into different components.

EXERCISE 1

1. Client server is modular infrastructure, this is intended to improve Usability, Flexibility, Interoperability and Scalability. Explain each with an example, in each case how it helps to improve the functionality of client server architecture.
2. Explain the following.
   (*a*) Computing in client server architecture over Mainframe architecture has certain advantages and disadvantages. Describe atleast two advantages and disadvantages for each architecture.
   (*b*) Client/Server architecture could be explained as 2-tier architecture. Explain.
   (*c*) Explain the working of three-tier architecture with an application server.
   (*d*) How does the client server interaction work? Explain with a sketch.
   (*e*) Describe the server function and client responsibility in this architecture.
3. Differentiate between Stateful and Stateless servers.
4. Describe three-level schema architecture. Why do we need mapping between schema levels?
5. Differentiate between Transaction server and Data server system with example.
6. In client server architecture, what do you mean by Availability, Reliability, Serviceability and Security? Explain with examples.
7. How client/server computing environment is different from mainframe based computing environment?
8. In the online transaction processing environment, discuss how transaction processing monitor controls data transfer between client and server machines.

This page intentionally left blank

2

Driving Forces Behind Client/ Server Computing

2.1 INTRODUCTION

A rapidly changing business environment generates a demand for enterprise – wide data access, which, in turn, sets the stage for end user productivity gains. Data access requirements have given rise to an environment in which computers work together to form a system, often called distributed computing, cooperative computing, and the like.

To be competitive in a global economy, organizations in developed economies must employ technology to gain the efficiency necessary to offset their higher labour costs. Re-engineering the business process to provide information and decision-making support at points of customer contact reduces the need for layers of decision-making management, improves responsiveness, and enhance customer service. Empowerment means that knowledge and responsibility are available to the employee at the point of customer contact. Empowerment will ensure that product and services problems and opportunities are identified and centralized. Client/Server computing is the most effective source for the tools that empower employees with authority and responsibility.

However, Client/Server computing has become more practical and cost-effective because of changes in computer technology that allow the use of PC-based platforms with reliability and robustness comparable to those of traditional mainframe system. In fact, the accelerating trend toward system development based on Internet Technologies, particularly those supplied by Web, has extended the Client/Server model's reach and relevance considerably. For example, to remain competitive in a global business environment, businesses are increasingly dependent on the Web to conduct their marketing and service operations. Such Web-based electronic commerce, known as E-commerce, is very likely to become the business norm for businesses of all sizes.

Even a cursory examination of Websites will demonstrate the Web's search. Organizations that range in size from Microsoft, IBM, GM, and Boeing to local arts/craft and flower shops

conduct part – or even most – of their business operations via E-commerce. There are various forces that drive the move to client/server computing. Some of them are:

(*i*) The changing business environment.
(*ii*) Globalization: The world as a market.
(*iii*) The growing need for enterprise data access.
(*iv*) The demand for end user productivity gains based on the efficient use of data resources.
(*v*) Technological advances that have made client/server computing practical like microprocessor technology, data communication and Internet, Database systems, Operating Systems and Graphical User Interface, PC-based and end user application software.
(*vi*) Growing cost and performance advantages of PC-based platforms.
(*vii*) Enterprise network management.

2.2 DRIVING FORCES

Forces that drives the move to Client/Server computing widely can be classified in two general categories based on:

(*i*) Business perspective.
(*ii*) Technology perspective.

2.2.1 Business Perspective

Basically the business perspective should be kept in mind for obtaining the following achievements through the system:

- For increased productivity.
- Superior quality.
- Improved responsiveness.
- Focus on core business.

The effective factors that govern the driving forces are given below:

The changing business environment: Business process engineering has become necessary for competitiveness in the market which is forcing organizations to find new ways to manage their business, despite fewer personnel, more outsourcing, a market driven orientation, and rapid product obsolescence.

Due to globalization of business, the organizations have to meet global competitive pressure by streamlining their operations and by providing an ever-expanding array of customer services. Information management has become a critical issue in this competitive environment; marketing fast, efficient, and widespread data access has become the key to survival. The corporate database has become a far more dynamic asset than it used to be,

and it must be available at relatively low cost. Unfortunately, the demand for a more accessible database is not well-served by traditional methods and platforms. The dynamic information driven corporate worlds of today require data to be available to decision makers on time and in an appropriate format. Because end users have become active in handling their own basic data management and data analysis, the movement towards freedom of data access has made Client/Server computing almost inevitable.

One might be tempted to urge that microcomputer networks constitute a sufficient answer to the challenge of dynamic data access. Unfortunately, even the use of networks that tie legions of PC's together is an unsatisfactory solution if request processing overloads the network. The Client/Server model's ability to share resources efficiently by splitting data processing yields a more efficient utilization of those resources. It is not surprising that Client/Server computing has received so much attention from such a wide spectrum of interested parties.

Globalization

Conceptually, the world has begun to be treated as a market. Information Technology plays an important role in bringing all the trade on a single platform by eliminating the barriers. IT helps and supports various marketing priorities like quality, cost, product differentiation and services.

The growing need for enterprise data access: One of the major MIS functions is to provide quick and accurate data access for decision- making at many organizational levels. Managers and decision makers need fast on-demand data access through easy-to-use interfaces. When corporations grow, and especially when they grow by merging with other corporations, it is common to find a mixture of disparate data sources in their systems. For example, data may be located in flat files, in hierarchical or network databases or in relational databases. Given such a multiple source data environment, MIS department managers often find it difficult to provide tools for integrating and aggregating data for decision-making purposes, thus limiting the use of data as a company asset. Client server computing makes it possible to mix and match data as well as hardware. In addition, given the rapidly increasing internet-enabled access to external data through the Internet's inherent Client/Server architecture, corporate Client/Server computing makes it relatively easy to mix external and internal data.

The demand for end user productivity gains based on the efficient use of data resources: The growth of personal computers is a direct result of the productivity gains experienced by end-users at all business levels. End user demand for better ad hoc data access and data manipulation, better user interface, and better computer integration helped the PC gain corporate acceptance. With sophisticated yet easy to use PCs and application software, end user focus changed from how to access the data to how to manipulate the data to obtain information that leads to competitive advantages.

2.2.2 Technology Perspective

Technological advances that have made Client/Server computing practical by proper use of the following:

- Intelligent desktop devices.
- Computer network architectures.
- Technical advances like microprocessor technology, data communication and Internet Database system, operating system and graphical user interface.
- Trends in computer usage like:

(*i*) Standardization: Trend towards open systems and adaptation of industry standards, which includes:

* *de facto* standard: protocol or interface that is made public & widely accepted. (e.g., SNA, TCP/IP, VGA)
* *de jure* standard: protocol or interface specified by a formal standards making body. (e.g., ISO's OSI, ANSI C)

(*ii*) Human-Computer Interaction (HCI): trend towards GUI, user Control.

(*iii*) Information dissemination: trend towards data warehousing, data mining.

- PC-based end user application software together with the increasing power and capacity of workstations.
- Growing cost and performance are advantages of PC-based platforms.

The PC platform often offers unbeatable price/performance ratio compared to mainframe and minicomputer platforms. PC application cost, including acquisition, installation, training, and use, are usually lower than those of similar minicomputer and mainframe applications. New PC-based software makes use of very sophisticated technologies, such as object orientation, messaging, and tele-communications. These new technologies make end users more productive by enabling them to perform very sophisticated tasks easily, quickly, and efficiently. The growing software sophistication even makes it possible to migrate many mission-critical applications to PCs.

The pursuit of mainframe solutions typically means high acquisition and maintenance costs, and chances are that managers are locked into services provided by single source. In contrast, PC hardware and software costs have both declined sharply during the past few years. PC-based solutions typically are provided by many sources, thus limiting single-source vulnerability. However, multi-source solutions can also become a major management headache when system problems occur.

Enterprise Computing and the Network Management

If a business is run from its distributed locations, the technology supporting these units must be as reliable as the existing central systems. Technology for remote management of the distributed technology is essential in order to use scarce expertise appropriately and to reduce costs.

All computing and communications resources are integrated functionally as a single, seamless system. To maximize productivity by providing universal, up-to-date information the technology requirements are that computing technology must be widely deployed. All computers must be networked together in a consistent architecture such that computing and networking resources must be reliable, secure, and capable of delivering accurate information in a timely manner. Maximum capture of information relating to the business and its customers must occur within every business process. That information must be normalized, within reach of all users. To achieve that, mechanics employed to locate, access the data and also for hiding the transmit data. And all the applications must be flexible to user preferences and work styles i.e., applications must interwork with in a common framework.

Client/server technology gives cost-effective, logical, and consistent architectural model for networking that generalizes the typical computer model. Client/Server can simplify network interactions that will give transparent interaction to the users. See the Fig. 2.1 illustrated below:

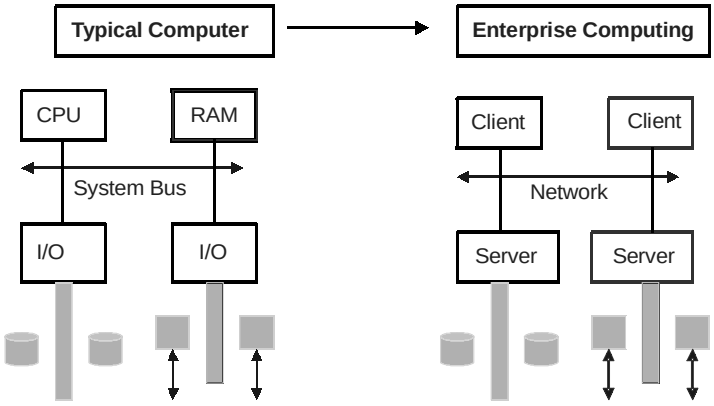


Fig. 2.1: Enterprise Computing

2.3 DEVELOPMENT OF CLIENT/SERVER SYSTEMS

The development of Client/Server systems differs greatly in process and style from the traditional information systems development methods. For example, the systems development approach, oriented towards the centralized mainframe environment and based on traditional programming language, can hardly be expected to function well in a client server environment that is based on hardware and software diversity. In addition a modern end users are more demanding and are likely to know more about computer technology than users did before the PC made its inroads. Then the concerning manager should pertain their knowledge about new technologies that are based on multiple platforms, multiple GUIs, multiple network protocols, and so on.

2.3.1 Development Tools

In today's rapid changing environment, choosing the right tools to develop Client/Server applications is one of the most critical decisions. As a rule of thumb, managers tend to choose a tool that has a long-term survival potential. However, the selection of a design or application development tool must also be driven by system development requirements. Once such requirements have been delineated, it is appropriate to determine the characteristics of the tool that you would like to have. Client/Server tools include:

- GUI-based development.
- A GUI builder that supports multiple interfaces (Windows, OS/2, Motif, Macintosh, and so on).
- Object-oriented development with a central repository for data and applications.
- Support for multiple database (flat file, hierarchical, networked, relational).
- Data access regardless of data model (using SQL or native navigational access).
- Seamless access to multiple databases.
- Complete SDLC (System Development Life Cycle) support from planning to implementation and maintenance.
- Team development support.
- Support for third party development tools (CASE, libraries, and so on)
- Prototyping and Rapid Application Development (RAD) capabilities.
- Support for multiple platforms (OS, Hardware, and GUIs).
- Support for middle ware protocols (ODBC, IDAPI, APPC, and so on).
- Multiple network protocol support (TCP/IP, IXP/SPX, NetBIOS, and so on).

There is no single best choice for any application development tool. For one thing, not all tools will support all the GUI's, operating system, middleware, and databases. Managers must choose a tool that fits the application development requirements and that matches the available human resources, as well as the hardware infrastructure. Chances are that the system will require multiple tools to make sure that all or most of the requirements are met. Selecting the development tools is just one step. Making sure that the system meets its objectives at the client, server, and network level is another issue.

2.3.2 Development Phases

It is important that a marketing plan be developed before actually starting the design and development efforts. The objective of this plan is to build and obtain end user and managerial support for the future Client/Server environment. Although there is no single recipe for this process, the overall idea is to conceptualize Client/Server system in terms of their scope, optimization of resources and managerial benefits. In short, the plan requires an integrated effort across all the departments within an organization. There are six main phases in Client/Server system development.

(*i*) Information System Infrastructure Self-study

The objective is to determine the actual state of the available computer resources. The self-study will generate atleast the following.

- A software and hardware inventory.
- A detailed and descriptive list of critical applications.
- A detailed human resource (personal and skills) inventory.
- A detailed list of problems and opportunities.

(*ii*) Client/Server Infrastructure Definition

The output of Phase One, combined with the company's computer infrastructure goal, is the input for the design of the basic Client/Server infrastructure blueprint. This blue print will address the main hardware and software issues for the client, server, and networking platforms.

(*iii*) Selecting a Window of Opportunity

The next stage is to find the right system on which to base the Client/Server pilot project. After identifying the pilot project, we need to define it very carefully by concentrating on the problem, available resources, and set of clearly defined and realistic goals. The project is described in business terms rather than technological jargon. When defining the system, we must make sure to plan for cost carefully. We should try to balance the cost carefully with the effective benefits of the system. We should also make sure to select a pilot implementation that provides immediate and tangible benefits. For example, a system that takes two years to develop and another three to generate tangible benefits is not acceptable.

(*iv*) Management Commitment

Top to bottom commitment is essential when we are dealing with the introduction of new technologies that affect the entire organization. We also need managerial commitment to ensure that the necessary resources (people, hardware, software, money, infrastructure) will be available and dedicated to the system. A common practice is to designate a person to work as a guide, or an agent of change, within the organization's departments. The main role of this person is to ease the process that changes people's role within the organization.

(*v*) Implementation

Guidelines to implementation should atleast include:

- Use "open" tools or standard-based tools.
- Foster continuing education in hardware, software, tools, and development principles.
- Look for vendors and consultants to provide specific training and implementation of designs, hardware, application software.

(*vi*) Review and Evaluation

We should make sure that the system conforms to the criteria defined in Phase Three. We should continuously measure system performance as the system load increases, because typical Client/Server solutions tend to increase the network traffic and slow down the

network. Careful network performance modelling is required to ensure that the system performs well under heavy end user demand conditions. Such performance modeling should be done at the server end, the client end, and the network layer.

2.4 CLIENT/SERVER STANDARDS

Standards assure that dissimilar computers, networks, and applications scan interact to form a system. But what constitutes standards? A standard is a publicly defined method to accomplish specific tasks or purposes within a given discipline and technology. Standards make networks practical.

Open systems and Client-Server computing are often used as if they were synonymous. It does not make long-term sense for users to adopt a Client/Server environment that is not based on standards. There are currently very few Client/Server technologies based on standards at every level. Proprietary Client/Server technologies (applications, middleware etc.) will always lock you into a particular supplier. The existing costs are always high. Failure to appreciate the spectrum of technologies within the Client-Server model, will always lead to dysfunctional Client/Server solutions. This will result in compromises in key areas of any company's Client/Server infrastructure, such as Usability, Security, and Performance.

There are quite a few organizations whose members work to establish the standards that govern specific activities. For example, the Institute of Electrical and Electronics Engineers (IEEE) are dedicated to define the standards in the network hardware environment. Similarly, the American National Standards Institute (ANSI) has created standards for programming languages such as COBOL and SQL. The International Organization for Standardization (ISO) produces the Open System Interconnection (OSI) reference model to achieve network systems communications compatibility.

Benefits of Open Standards

- Standards allow us to incorporate new products and technology with existing I.T. investments — hardware, operating environments, and training, with minimum effort.
- Standards allow us to mix and match the 'best of breed' products. Thus databases and development tools, and Connectivity software become totally independent.
- Standards allow us to develop modular applications that do not fall apart because the network has been re-configured (e.g., change of topology, or transport protocol etc.), or the graphical user interface standard as changed, or a component-operating environment has changed.
- Standards maintain tighter security.
- Standards reduce the burden of overall maintenance and system administration.
- Standards provide faster execution of pre-compiled code.
- Standards prevent the database and its application and possibly others on the server from having their response time degraded in a production environment by inefficient queries.

2.5 CLIENT/SERVER SECURITY

A security threat is defined as circumstance, condition, or event with the potential to cause economic hardship to data or network resources in the form of destruction. Disclosure, modification of data, denial of service, and/or fraud, waist and abuse. Client/Server security issues deal with various authorization methods related to access control. Such mechanisms include password protection, encrypted smart cards. Biometrics and firewalls. Client/Server security problems can be due to following:

- **Physical security holes:** These results when any individual gains unauthorized access to a computer by getting some user's password.
- **Software security holes:** These result due to some bug in the software, due to which the system may be compromised into giving wrong performance.
- **Inconsistent usage holes:** These may result when two different usages of a systems contradict over a security point.

Of the above three, software security holes and inconsistent usage holes can be eliminated by careful design and implementation. For the physical security holes, we can employ various protection methods. These security methods can be classified into following categories:

(*i*) Trust-based security.
(*ii*) Security through obscurity.
(*iii*) Password scheme.
(*iv*) Biometric system.

2.5.1 Emerging Client/Server Security Threats

We can identify emerging Client/Server security threats as:

(*i*) Threats to local computing environment from mobile code,
(*ii*) Threats to servers that include impersonation, eavesdropping, denial of service, packet reply, and packet modification.

Software Agents and the Malicious Code Threat

Software agents or mobile code are executable programs that have ability to move from machine to machine and also to invoke itself without external influence. Client threats mostly arise from malicious data or code. Malicious codes refers to viruses, worms (a self-replicating program that is self-contained and does not require a host program. The program creates a copy of itself and causes it to execute without any user intervention, commonly utilizing network services to propagate to other host systems.)e.g., Trojan horse, logic bomb, and other deviant software programs. Virus is a code segment that replicates by attaching copies of itself to existing executables. The new copy of the virus is executed when a user executes the host programs. The virus may get activated upon the fulfilment of some specific conditions.

The protection method is to scan for malicious data and program fragments that are transferred from the server to the client, and filter out data and programs known to be dangerous.

2.5.2 Threats to Server

Threats to server may be of the following types:

(*i*) Eavesdropping is the activity of silently listening to the data sent over the network. This often allows a hacker to make complete transcript of network activity and thus obtain sensitive information, such as password, data, and procedures for performing functions. Encryption can prevent eavesdroppers from obtaining data traveling over unsecured networks.

(*ii*) Denial of service is a situation, where a user renders the system unusable for legitimate users by hogging or damaging a resource so that it can be used. The common forms of this, are:

- **Service overloading:** A server may be rendered useless by sending it a large amount of illegitimate service requests so as to consume up its CPU cycle resource. In such a situation, the server may deny the service request of legitimate requests.
- **Message flooding:** It is a process of increasing the number of receiving processes running over the disk of the server by sending large files repeatedly after short intervals. This may cause disk crash.
- **Packet replay** refers to the recording and retransmission of message packets in the network. Medium tapping can do this. A checker may gain access to a secure system by recording and later replaying a legitimate authentication sequence message. Packet reply can also be used to distort the original message. Using a method like packet time stamping and sequence counting can prevent this problem.

2.6 ORGANIZATIONAL EXPECTATIONS

As we have already discussed the advantages and disadvantages associated with Client/Server computing, from the organizational point of view the managers are looking for the following Client/Server benefits.

- Flexibility and adaptability.
- Improved employee productivity.
- Improved company work flow and a way to re-engineering business operations.
- New opportunities to provide competitive advantages.
- Increased customer service satisfaction.

Flexibility and Adaptability

Client/Server computing is expected to provide necessary organizational flexibility to adapt quickly and efficiently in changing business conditions. Such changes can be driven by technological advantages; government regulations, mergers and acquisitions, market forces and so on. A company that can adapt quickly to changes in its market conditions is more likely to survive than one that cannot.

Multinational companies, whose widely dispersed offices must share information across often-disparate computer platforms, are especially well-positioned to benefit from the flexibility and adaptability offered by the Client/Server infrastructure.

Improved Employee Productivity

Client/Server computing opens the door to previously unavailable corporate data. End users can manipulate and analyze such data on an ad hoc basis by means of the hardware and the software tools that are commonly available with client server environments. Quick and reliable information access enables end users to make intelligent decisions. Consequently, end users are more likely to perform their jobs better, provide better services, and become more productive within the corporation.

Improved Company Work Flow and a Way to Re-engineering Business Operations

Organizations that face problems with their internal data management typically favour the introduction of Client/Server computing. Providing data access is just the first step in information management. Providing the right data to the right people at the right time is the core of decision support for MIS departments. As competitive conditions change, so do the companies' internal structure, thus triggering demands for information systems that reflect those changes. Client/Server tools such as Lotus Notes are designed exclusively to provide corporations with data and forms distribution, and work group support, without regard to geographical boundaries. These workgroup tools are used to route the forms and data to the appropriate end users and coordinate employee work. The existence and effective use of such tools allows companies to re-engineer their operational processes, effectively changing the way they do the business.

New Opportunities to Provide Competitive Advantages

New strategic opportunities are likely to be identified as organizations restructure. By making use of such opportunities, organizations enhance their ability to compete by increasing market share through the provision of unique products or services. Proper information management is crucial within such a dynamic competitive arena. Therefore, improved information management provided by a Client/Server system means that such systems could become effective corporate strategic weapons.

Increased Customer Service Satisfaction

As new and better services are provided, customer satisfaction is likely to improve. Client/Server systems enable the corporate MIS manager to locate data closer to the source of data demand, thus increasing the efficiency with which customer enquiries are handled.

2.7 IMPROVING PERFORMANCE OF CLIENT/SERVER APPLICATIONS

Client/Server-developed applications may achieve substantially greater performance when compared with traditional workstations or host-only applications.

(*i*) Offload work to server: Database and communications processing are frequently offloaded to a faster server processor. Some applications processing also may be offloaded, particularly for a complex process, which is required by many users. The advantage of offloading is realized when the processing power of the server is significantly greater than that of the client workstation. Separate processors best support shared databases or specialized communications interfaces. Thus, the client workstation is available to handle other client tasks. These advantages are best realized when the client workstation supports multitasking or atleast easy and rapid task switching.

(*ii*) Reduce total execution time: The server can perform database searches, extensive calculations, and stored procedure execution in parallel while the client workstation deals directly with the current user needs. Several servers can be used together, each performing a specific function. Servers may be multiprocessors with shared memory, which enables programs to overlap the LAN functions and database search functions. In general, the increased power of the server enables it to perform its functions faster than the client workstation. In order for this approach to reduce the total elapsed time, the additional time required to transmit the request over the network to the server must be less than the saving. High-speed local area network topologies operating at 4, 10, 16, or 100Mbps (megabits per second) provide high-speed communications to manage the extra traffic in less time than the savings realized from the server. The time to transmit the request to the server, execute the request, and transmit the result to the requestor, must be less than the time to perform the entire transaction on the client workstation.

(*iii*) Use a multitasking client: As workstation users become more sophisticated, the capability to be simultaneously involved in multiple processes becomes attractive. Independent tasks can be activated to manage communications processes, such as electronic mail, electronic feeds from news media and the stock exchange, and remote data collection (downloading from remote servers). Personal productivity applications, such as word processors, spreadsheets, and presentation graphics, can be active. Several of these applications can be dynamically linked together to provide the desktop information-processing environment. Functions such as Dynamic Data Exchange (DDE) and Object Linking and Embedding (OLE) permit including spreadsheets dynamically into word-processed documents. These links can be *hot* so that changes in the spreadsheet cause the word-processed document to be updated, or they can be *cut and paste* so that the current status of the spreadsheet is copied into the word-processed document.

Systems developers appreciate the capability to create, compile, link, and test programs in parallel. The complexity introduced by the integrated CASE environment requires multiple processes to be simultaneously active so the workstation need not be dedicated to a single long-running function. Effective use of modern CASE tools and workstation development products requires a client workstation that supports multitasking.

2.8 SINGLE SYSTEM IMAGE

Rapid changes have occurred in computer technology resulting in system of increased capabilities. This indicates that maximum resources are available to accepts all these new products. For the organizations using Client/Server systems the environment is heterogeneous whereas the users prime concern to achieve the maximum functionality. Every Client/Server system should give equal importance to the developers' and users' requirements. For the users, this means the realization of a single-system-image. "A single-system-image is the illusion, created by software or hardware, that presents a collection of resources as one, more powerful resource." SSI makes the system appear like a single machine to the user, to applications, and to the network. With it all network resources present themselves to every user in the same way from every workstation (See the Fig. 2.2, given below) and can be used transparently after the user has authorized himself/herself once. The user environment with a desktop and often-used tools, such as editors and mailer, is also organized in a uniform way. The workstation on the desk appears to provide all these services. In such an environment the user need not to bother about how the processors (both the client and the server) are working, where the data storage take place and which networking scheme has been selected to build the system.

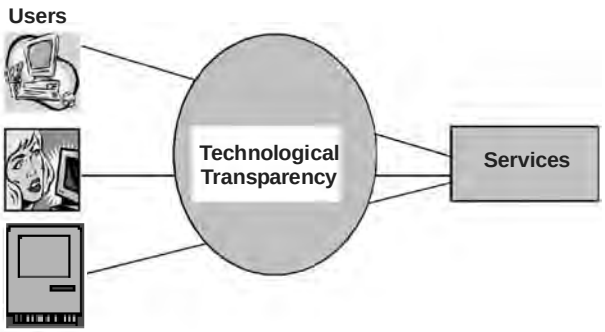


Fig.2.2: Single Image System

Further desired services in *single-system-image* environment are:

- Single File Hierarchy; for example: xFS, AFS, Solaris MC Proxy.
- Single Control Point: Management from single GUI and access to every resource is provided to each user as per their valid requirements.

- Single virtual networking.
- Single memory space e.g. Network RAM/DSM.
- Single Job Management e.g. Glunix, Codine, LSF.
- Single User Interface: Like workstation/PC windowing environment (CDE in Solaris/NT), Web technology can also be used.
- Standard security procedure: Access to every application is provided through a standard security procedure by maintaining a security layer.
- Every application helps in the same way to represent the errors and also to resolve them.
- Standard functions work in the same way so new applications can be added with minimal training. Emphasis is given on only new business functions.

Hence, *single-system-image* is the only way to achieve acceptable technological transparency.

"A single-system-image of all the organization's data and easy management of change" are the promises of client/server computing.

But as more companies follow the trend towards downsized Client/Server networks, some find the promise elusive. Security, scalability and administration costs are three of the key issues. For example, the simple addition of a new user can require the definition to be added to every server in the network. Some of the visible benefits due to *single-system-image* are as given below:

- Increase the utilization of system resources transparently.
- Facilitates process migration across workstations transparently along with load balancing.
- Provides improved reliability and higher availability.
- Provides overall improved system response time and performance.
- Gives simplified system management.
- Reduces the risk covered due to operator errors.
- User need not be aware of the underlying system.
- Provides such architecture to use these machines effectively.

2.9 DOWNSIZING AND RIGHTSIZING

Downsizing: The downward migrations of business applications are often from mainframes to PCs due to low costing of workstation. And also today's workstations are as powerful as last decade's mainframes. The result of that is Clients having power at the cost of less money, provides better performance and then system offers flexibility to make other purchase or to increase overall benefits.

Rightsizing: Moves the Client/Server applications to the most appropriate server platform, in that case the servers from different vendors can co-exist and the network is known as the 'system'. Getting the data from the system no longer refers to a single mainframe. As a matter of fact, we probably don't know where the server physically resides.

Upsizing: The bottom-up trend of networking all the stand alone PCs and workstations at the department or work group level. Early LANs were implemented to share hardware (printers, scanners, etc.). But now LANs are being implemented to share data and applications in addition to hardware.

Mainframes are being replaced by lesser expensive PC's on networks. This is called computer downsizing. Companies implementing business process reengineering are downsizing organizationally. This is called business downsizing. All this would result in hundreds of smaller systems, all communicating to each other and serving the need of local teams as well as individuals working in an organization. This is called cultural downsizing. The net result is distributed computer systems that support decentralized decision-making. This is the client/server revolution of the nineties.

2.10 CLIENT/SERVER METHODOLOGY

Many PC-based developers, particularly those who never knew of any other type of computer, believe that today's methodologies are not only wrong, but also unnecessary. They believe that prototyping based on rapid application development tools make methodologies completely unnecessary. Is this true? If yes, should the methodologies be thrown away? The answer to all these questions depends on the scale and complexity of the application being developed. Small applications that run on a single desktop can be built within hours. The use of methodology in such cases can be waste of time.

However, bigger systems are qualitatively different, especially in term of their design process. Whenever, a system, particularly one involving a database, expands to include more than one server, with servers being located in more than one geographical location, complexity is bound to go up. Distributed systems cross this complexity barrier rapidly. We can say.

- Methodologies are important, and will continue to remain so for the construction of large applications.
- Distributed systems will need these methodologies most of all.
- Today's methodologies will have to change to meet the needs of a new generation of developers and users, accommodate the design of distributed systems, and yield friendly, maintainable systems.

EXERCISE 2

1. Explain various Clients/Server system development tools.
2. Write short notes on the following.
   (*a*) Single system image.
   (*b*) Downsizing and Client/Server computing.
3. Explain Client/Server System development methodology and explain various phases and their activities involved in System Integration Life Cycle (SILC).
4. Explain the following in detail:-
   (*a*) Performance evaluation of Client/Server Application.
   (*b*) Reliability and Serviceability of Client/Server Architecture.
5. Differentiate between Downsizing and Client/Server Computing.
6. Explain different ways to improve performance in Client/Server developed applications.
7. What is client server system development methodology? Explain different phases of System Integration Life-Cycle.
8. How are software's distributed in client server model? In the client server environment, what are performance- monitoring tools for different operating system?
9. What are the various ways to reduce network traffic of client server computing?

3

Architectures of Client/Server Systems

3.1 INTRODUCTION

The term Client/Server was first used in the 1980s in reference to personal computers (PCs) on a network. The actual Client/Server model started gaining acceptance in the late 1980s. The term Client/Server is in reality a logical concept. The client and server components may not exist on distinct physical hardware. A single machine can be both a client and a server depending on the software configuration. The Client/Server technology is a model, for the interaction between simultaneously executing software processes. The term architecture refers to the logical structure and functional characteristics of a system, including the way they interact with each other in terms of computer hardware, software and the links between them.

In case of Client/Server systems, the architecture means the way clients and servers along with the requisite software are configured with each others. Client/Server architecture is based on the hardware and the software components that interact to form a system. The limitations of file sharing architectures led to the emergence of the Client/Server architecture. This approach introduced a database server to replace the file server. Using a Relational Database Management System (RDBMS), user queries could be answered directly. The Client/Server architecture reduced network traffic by providing a query response rather than total file transfer. It improves multi-user updating through a GUI front-end to a shared database. In Client/Server architectures, Remote Procedure Calls (RPCs) or Structural Query Language (SQL) statements are typically used to communicate between the client and server.

File sharing architecture (not a Client/Server architecture):

File based database (flat-file database are very efficient to extracting information from large data files. Each workstation on the network has access to a central file server where the data is stored.

The data files can also reside on the workstation with the client application. Multiple workstations will access the same file server where the data is stored. The file server is centrally located so that it can be reached easily and efficiently by all workstations.

The original PC networks were based on file sharing architectures, where the server downloads files form the shared location to the desktop environment. The requested user job is then run (including logic and data) in the desktop environment.

File sharing architectures work if shared usage is low, update contention is low, and the volume of data to be transferred is low. In the 1990s, PC LAN (Local Area Network) computing changed because the capacity of file sharing was strained as the number of online users grew (it can only satisfy about 12 users simultaneously) and Graphical User Interfaces (GUIs)became popular (making mainframe and terminal displays appear out of data). PCs are now being used in Client/Server architectures.

Mainframe architecture (not a Client/Server architecture)

With mainframe software architectures all intelligence is within the central host computer. Users interact with the host through a terminal that captures keystrokes and sends that information to the host. Mainframe software architectures are not tied to a hardware platform. User interaction can be done using PCs and UNIX workstations. A limitation of mainframe software architectures is that they do not easily support graphical user interfaces or access to multiple databases from geographically dispersed sites. In the last few years, mainframes have found a new use as a server in distributed Client/Server architectures.

The Client/Server software architecture is a versatile, message-based and modular infrastructure that is intended to improve usability, flexibility, interoperability, and scalability as compared to centralized, mainframe, time sharing computing.

3.2 COMPONENTS

Client/Server architecture is based on hardware and software components that interact to form a system. The system includes mainly three components.

(*i*) Hardware (client and server).

(*ii*) Software (which make hardware operational).

(*iii*) Communication middleware. (associated with a network which are used to link the hardware and software).

The client is any computer **process** that requests services from server. The client uses the services provided by one or more server processors. The client is also known as the **front-end application**, reflecting that the end user usually interacts with the client process.

The server is any computer **process** providing the services to the client and also supports multiple and simultaneous clients requests. The server is also known as **back-end application**, reflecting the fact that the server process provides the background services for the client process.

The communication middleware is any computer **process through** which client and server communicate. Middleware is used to integrate application programs and other software components in a distributed environment. Also known as **communication layer**. Communication layer is made up of several layers of software that aids the transmission of data and control information between Client and Server. Communication middleware is usually associated with a network. The Fig. 3.1 below gives a general structure of Client/ Server System.

Now as the definition reveals, clients are treated as the front-end application and the server as the back-end application, the Fig. 3.2 given below shows the front-end and back-end functionality.

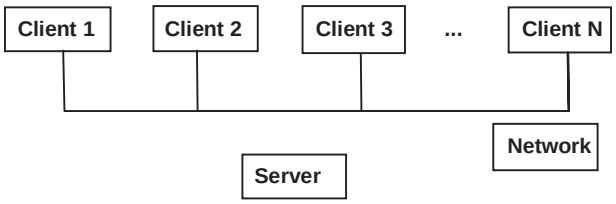


Fig.3.1: Structure of a Client/Server System

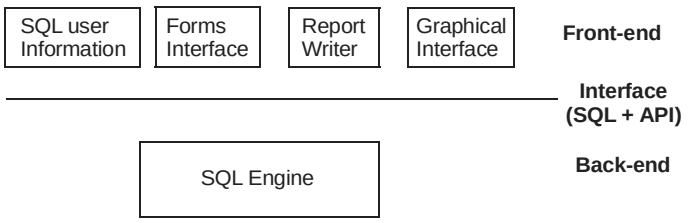


Fig.3.2: Front-end and Back-end Functionality

3.2.1 Interaction between the Components

The interaction mechanism between the components of Client/Server architecture is clear from the Fig. 3.3. The client process is providing the interface to the end users. Communication middleware is providing all the possible support for the communication taking place between the client and server processes. Communication middleware ensures that the messages between clients and servers are properly routed and delivered. Requests are handled by the database server, which checks the validity of the request, executes them, and send the result back to the clients.

3.2.2 Complex Client/Server Interactions

The better understanding about the functionality of Client/Server is observed when the clients and server interact with each other. Some noticeable facts are:

- A client application is not restricted to accessing a single service. The client contacts a different server (perhaps on a different computer) for each service.
- A client application is not restricted to accessing a single server for a given service.

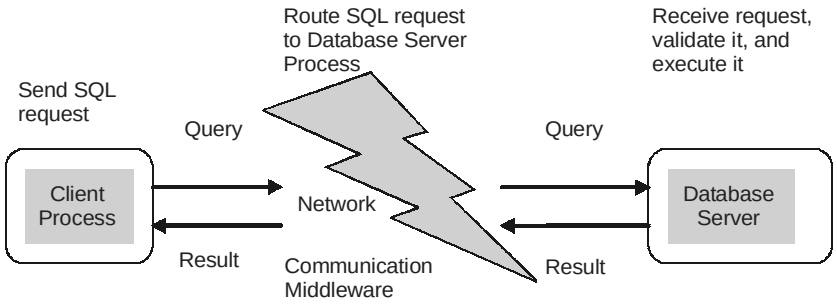


Fig.3.3: Components Interaction

- A server is not restricted from performing further Client/Server interactions — a server for one service can become a client of another.

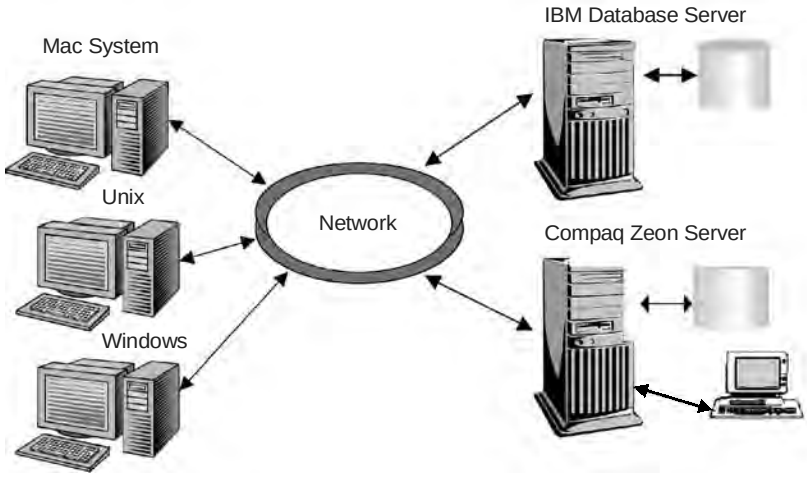


Fig.3.4: A Complex Client/Server Environment

Generally, the client and server processes reside on different computers. The Fig. 3.4 illustrates a Client/Server system with more than one server and several clients. The system comprises of the Back-end, Front-end Processes and Middleware.

Back-end processes as: IBM Database server process and Compaq Zeon Server

Front-end as: Application client processes (Windows, Unix and Mac Systems)

Middleware as: Communication middleware (network and supporting software)

The client process runs under different Operating Systems (Windows, Unix and Mac System), server process (IBM and Compaq computers) runs under different operating system

(OS/2 and Unix). The communication middleware acts as the integrating platform for all the different components. The communication can take place between client to client and as well as server to server.

3.3 PRINCIPLES BEHIND CLIENT/SERVER SYSTEMS

The components of the Client/Server architecture must conform to some basic principles, if they are to interact properly. These principles must be uniformly applicable to client, server, and to communication middleware components. Generally, these principles generating the Client/Server architecture constitute the foundation on which most current-generation Client/Server system are built. Some of the main principles are as follows:

(*i*) Hardware independence.
(*ii*) Software independence.
(*iii*) Open access to services.
(*iv*) Process distribution.
(*v*) Standards.

(*i*) Hardware independence: The principles of hardware independence requires that the Client, Server, and communication middleware, processes run on multiple hardware platforms (IBM, DEC, Compaq, Apple, and so on) without any functional differences.

(*ii*) Software independence: The principles of software independence requires that the Client, Server, and communication middleware processes support multiple operating systems (such as Windows 98, Windows NT, Apple Mac system, OS/2, Linux, and Unix) multiple network protocols (such as IPX, and TCP/IP), and multiple application (spreadsheet, database electronic mail and so on).

(*iii*) Open access to services: All client in the system must have open (unrestricted) access to all the services provided within the network, and these services must not be dependent on the location of the client or the server. A key issue is that the services should be provided on demand to the client. In fact, the provision of on-demand service is one of the main objectives of Client/Server computing model.

(*iv*) Process distribution: A primary identifying characteristic of Client/Server system is that the processing of information is distributed among Clients and Servers. The division of the application-processing load must conform to the following rules:

- Client and server processes must be autonomous entities with clearly defined boundaries and functions. This property enables us to clearly define the functionality of each side, and it enhances the modularity and flexibility of the system.
- Local utilization of resources (at both client and server sides) must be maximized. The client and server process must fully utilize the processing power of the host computers. This property enables the system to assign functionality to the

computer best suited to the task. In other words, to best utilize all resources, the server process must be shared among all client processes; that is, a server process should service multiple requests from multiple clients.

- Scalability and flexibility requires that the client and server process be easily upgradeable to run on more powerful hardware and software platforms. This property extends the functionality of Client/Server processes when they are called upon to provide the additional capabilities or better performance.
- Interoperability and integration requires that client and server processes be seamlessly integrated to form a system. Swapping a server process must be transparent the client process.

(*v*) Standards: Now, finally all the principles that are formulated must be based on standards applied within the Client/Server architecture. For example, standard must govern the user interface, data access, network protocols, interprocess communications and so on. Standards ensure that all components interact in an orderly manner to achieve the desired results. There is no universal standard for all the components. The fact is that there are many different standards from which to choose. For example, an application can be based on Open Database Connectivity (ODBC) instead of Integrated Database Application Programming Interface (IDAPI) for Data access (ODBC and IDAPI are database middleware components that enables the system to provide a data access standard for multiple processes.) Or the application might use Internet work Packet Exchange (IPX) instead of Transmission Control Protocol/Internet Protocol (TCP/IP) as the network protocol. The fact that the application does not use single standards does not mean that it will be a Client/Server application. The point is to ensure that all components (server, clients, and communication middleware) are able to interact as long as they use the same standards. What really defines Client/Server computing is that the splitting of the application processing is independent of the network protocols used.

3.4 CLIENT COMPONENTS

As we know, the client is any process that requests services from the server process. The client is proactive and will, therefore, always initiate the conversation with the server. The client includes the software and hardware components. The desirable client software and hardware feature are:

(i) Powerful hardware.

(ii) An operating system capable of multitasking.

(iii) Communication capabilities.

(iv) A graphical user interface (GUI).

(*i*) Powerful hardware: Because client processes typically requires a lot of hardware resources, they should be stationed on a computer with sufficient computing power, such as fast Pentium II, III, or RISC workstations. Such processing power

facilitates the creation of systems with multimedia capabilities. A Multimedia system handles multiple data types, such as voice, image, video, and so on. Client processes also require large amount of hard disk space and physical memory, the more such a resource is available, the better.

(*ii*) An operating system capable of multitasking: The client should have access to an operating system with at least some multitasking capabilities. Microsoft Windows 98 and XP are currently the most common client platforms. Windows 98 and XP provide access to memory, pre-emptive multitasking capabilities, and a graphical user interface, which makes windows the platform of choice in a majority of Client/Server implementations. However, Windows NT, Windows 2000 server, OS/2 from IBM corporation, and the many "flavours" of UNIX, including Linux are well-suited to handle the Client/Server processing that is largely done at the server side of the Client/Server equation.

(*iii*) Communication capabilities: To interact efficiently in a Client/Server environment, the client computer must be able to connect and communicate with the other components in a network environment. Therefore, the combination of hardware and operating system must also provide adequate connectivity to multiple network operating systems. The reason for requiring a client computer to be capable of connecting and accessing multiple network operating systems is simple services may be located in different networks.

(*iv*) A graphical user interface (GUI): The client application, or front-end, runs on top of the operating system and connects with the communication middleware to access services available in the network. Several third generation programming languages (3GLs) and fourth generation languages (4GLs) can be used to create the front-end application. Most front-end applications are GUI-based to hide the complexity of the Client/Server components from the end user. The Fig. 3.5 given below illustrates the basic client components.

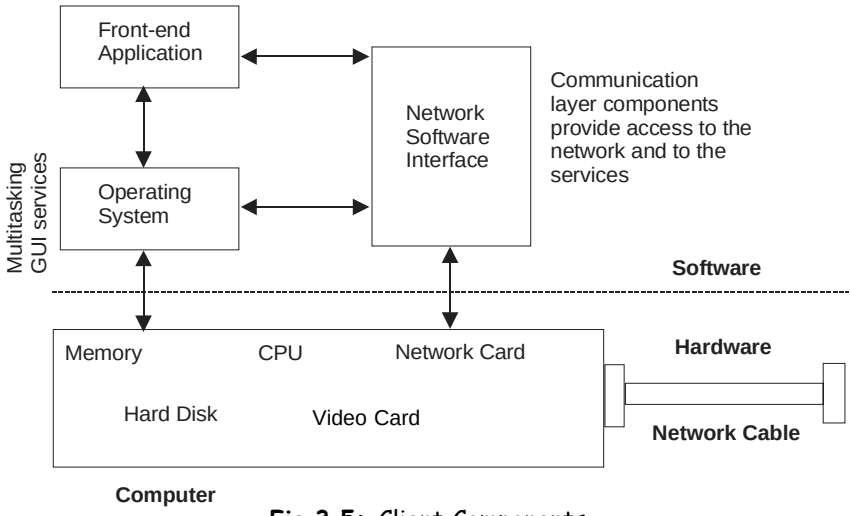


Fig.3.5: *Client Components*

3.5 SERVER COMPONENTS

As we have already discussed, the server is any process that provides services to the client process. The server is active because it always waits for the client's request. The services provided by server are:

(*i*) **File services**: For a LAN environment in which a computer with a big, fast hard disk is shared among different users, a client connected to the network can store files on the file server as if it were another local hard disk.

(*ii*) **Print services**: For a LAN environment in which a PC with one or more printers attached is shared among several clients, a client can access any one of the printers as if it were directly attached to its own computer. The data to be printed travel from the client's PC to the server printer PC where they are temporarily stored on the hard disk. When the client finishes the printing job, the data is moved from the hard disk on the print server to the appropriate printer.

(*iii*) **Fax services:** This requires at least one server equipped (internally or externally) with a fax device. The client PC need not have a fax or even a phone line connection. Instead, the client submits the data to be faxed to the fax server with the required information; such as the fax number or name of the receiver. The fax server will schedule the fax, dial the fax number, and transmit the fax. The fax server should also be able to handle any problems derived from the process.

(*iv*) **Communication services**: That let the client PCs connected to the communications server access other host computers or services to which the client is not directly connected. For example, a communication server allows a client PC to dial out to access board, a remote LA location, and so on.

(*v*) **Database services:** Which constitute the most common and most successful Client/Server implementation. Given the existence of database server, the client sends SQL request to the server. The server receives the SLQ code, validates it, executes it, and send only the result to the client. The data and the database engine are located in the database server computer.

(*vi*) **Transaction services:** Which are provided by transaction servers that are connected to the database server. A transaction server contains the database transaction code or procedures that manipulate the data in database. A front-end application in a client computer sends a request to the transaction server to execute a specific procedure store on the database server. No SQL code travels through the network. Transaction servers reduce network traffic and provide better performance than database servers.

(*vii*) **Groupware services:** Liable to store semi-structured information like Text, image, mail, bulletin boards, flow of work. Groupware Server provides services, which put people in contact with other people, that is because "groupware" is an ill-defined classification. Protocols differ from product to product. For examples: Lotus Notes/Domino, Microsoft Exchange.

(*viii*) Object application services: Communicating distributed objects reside on server. Object server provides access to those objects from client objects. Object Application Servers are responsible for Sharing distributed objects across the network. Object Application Servers uses the protocols that are usually some kind of Object Request Broker (ORB). Each distributed object can have one or more remote method. ORB locates an instance of the object server class, invokes the requested method, and returns the results to the client object. Object Application Server provides an ORB and application servers to implement this.

(*ix*) Web application services: Some documents, data, etc., reside on web servers. Web application provides access to documents and other data. "Thin" clients typically use a web browser to request those documents. Such services provide the sharing of the documents across intranets, or across the Internet (or extranets). The most commonly used protocol is HTTP (Hyper Text Transport Protocol). Web application servers are now augmenting simple web servers.

(*x*) Miscellaneous services: These include CD-ROM, video card, backup, and so on. Like the client, the server also has hardware and software components. The hardware components include the computer, CPU, memory, hard disk, video card, network card, and so on. The computer that houses the server process should be the more powerful computer than the "average" client computer because the server process must be able to handle concurrent requests from multiple clients. The Fig. 3.6 illustrates the components of server.

The server application, or back-end, runs on the top of the operating system and interacts with the communication middleware components to "listen" for the client request for the services. Unlike the front-end client processes, the server process need not be GUI based. Keep in mind that back-end application interacts with operating system (network or stand alone) to access local resources (hard disk, memory, CPU cycle, and so on). The back-end server constantly "listens" for client requests. Once a request is received the server processes it locally. The server knows how to process the request; the client tells the server only what it needs do, not how to do it. When the request is met, the answer is sent back to the client through the communication middleware.

The server hardware characteristics depend upon the extent of the required services. For example, a database is to be used in a network of fifty clients may require a computer with the following minimum characteristic:

- Fast CPU (RISC, Pentium, Power PC, or multiprocessor)
- Fault tolerant capabilities:
  - Dual power supply to prevent power supply problem.
  - Standby power supply to protect against power line failure.

- Error checking and correcting (ECC) memory to protect against memory module failures.
- Redundant Array to Inexpensive Disk (RAID) to provide protection against physical hardware failures.

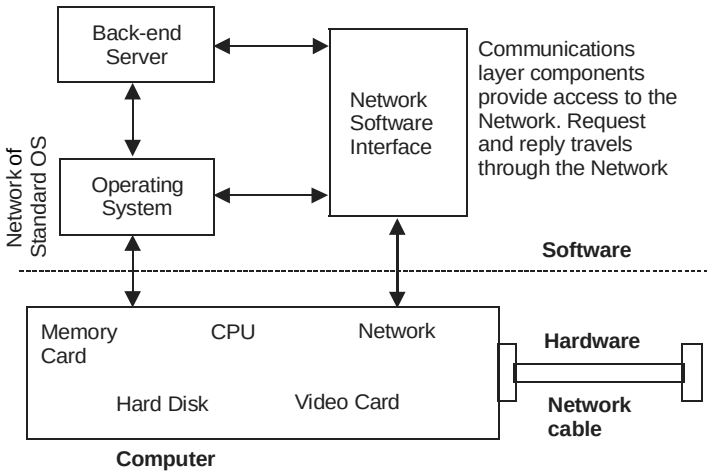


Fig.3.6: Server Components

- Expandability of CPU, memory disk, and peripherals.
- Bus support for multiple add-on boards.
- Multiple communication options.

In theory, any computer process that can be clearly divided into client and server components can be implemented through the Client/Server model. If properly implemented, the Client/Server architectural principles for process distribution are translated into the following server process benefits:

- **Location independence.** The server process can be located anywhere in the network.
- **Resource optimization.** The server process may be shared.
- **Scalability.** The server process can be upgraded to run on more powerful platforms.
- **Interoperability and integration.** The server process should be able to work in a "Plug and Play" environment.

These benefits added to hardware and software independence principles of the Client/ Server computing model, facilitate the integration of PCs, minicomputer, and mainframes in a nearly seamless environment.

3.5.1 The Complexity of Servers

The server processes one request at a time; we can say servers are fairly simple because they are sequential. After accepting a request, the server forms a reply and sends it before requesting to see if another request has arrived. Here, the operating system plays a big role in maintaining the request queue that arrives for a server.

Servers are usually much more difficult to build than clients because they need to accommodate multiple concurrent requests. Typically, servers have two parts:

- A single master program that is responsible for accepting new requests.
- A set of slaves that are responsible for handling individual requests.

Further, master server performs the following five steps (Server Functions):

(*i*) Open port: The master opens a port at which the client request reached.

(*ii*) Wait for client: The master waits for a new client to send a request.

(*iii*) Choose port: If necessary, the master allocates new local port for this request and informs the client.

(*iv*) Start slave: The master starts an independent, concurrent slave to handle this request (for example: in UNIX, it forks a copy of the server process). Note that the slave handles one request and then terminates–the slave does not wait for requests from other clients.

(*v*) Continue: The master returns to the wait step and continues accepting new requests while the newly created slave handles the previous request concurrently.

Because the master starts a slave for each new request, processing proceeds concurrently. In addition to the complexity that results because the server handles concurrent requests, complexity also arises because the server must enforce authorization and protection rules. Server programs usually need to execute with the highest privilege because they must read system files, keep logs, and access protected data. The operating system will not restrict a server program if it attempts to access a user files. Thus, servers cannot blindly honour requests from other sites. Instead, each server takes responsibility for enforcing the system access and protection policies.

Finally, servers must protect themselves against malformed request or against request that will cause the server program itself to abort. Often it is difficult to foresee potential problems. Once an abort occurs, no client would be able to access files until a system programmer restarts the server.

"Servers are usually more difficult to build than clients because, although they can be implemented with application programs, server must enforce all the access and protection policies of the computer system on which they run, and must protect themselves against all possible errors."

3.6 COMMUNICATIONS MIDDLEWARE COMPONENTS

The communication middleware software provides the means through which clients and servers communicate to perform specific actions. It also provides specialized services to the client process that insulates the front-end applications programmer from the internal working of the database server and network protocols. In the past, applications programmers had to write code that would directly interface with specific database language (generally a version of SQL) and the specific network protocol used by the database server. For example, when writing a front-end application to access an IBM OS/2 database manager database, the programmer had to write SQL and Net BIOS (Network Protocol) command in the application. The Net BIOS command would allow the client process to establish a session with the database server, send specific control information, send the request, and so on. If the same application is to be used with a different database and network, the application routines must be rewritten for the new database and network protocols. Clearly, such a condition is undesirable, and this is where middleware comes in handy. The definition of middleware is based on the intended goals and main functions of this new software category.

Although middleware can be used in different types of scenarios, such as e-mail, fax, or network protocol translation, most first generation middleware used in Client/Server applications is oriented toward providing transport data access to several database servers. The use of database middleware yields:

- **Network independence:** by allowing the front-end application to access data without regard to the network protocols.
- **Database server independence:** by allowing the front-end application to access data from multiple database servers without having to write code that is specific to each database server.

The use of database middleware, make it possible for the programmer to use the generic SQL sentences to access different and multiple database servers. The middleware layer isolates the program from the differences among SQL dialects by transforming the generic SQL sentences into the database server's expected syntax. For example, a problem in developing a front-end system for multiple database servers is that application programmers must have in-depth knowledge of the network communications and the database access language characteristic of each database to access remote data. The problem is aggravated by the fact that each DBMS vendor implements its own version of SQL (with difference in syntax, additional functions, and enhancement with respect to the SQL standard). Furthermore, the data may reside in a non-relational DBMS (hierarchical, network or flat files) that does not support SQL, thus making it harder for the programmers to access the data given such cumbersome requirements, programming in Client/Server systems becomes more difficult than programming in traditional mainframe system. Database middleware

eases the problem of accessing resources of data in multiple networks and releases the program from details of managing the network communications. To accomplish its functions, the communication middleware software operates at two levels:

- The physical level deals with the communications between client and server computers (computer to computer). In other words, it addresses how the computers are physically linked. The physical links include the network hardware and software. The network software includes the network protocol. Recall that network protocols are rules that govern how computers must interact with other computers in network, and they ensure that computers are able to send and receive signal to and from each other. Physically, the communication middleware is, in most cases, the network. Because the Client/Server model allows the client and server to reside on the same computer, it may exist without the benefit of a computer network.
- The logical level deals with the communications between client and server. Process (process to process) that is, with how the client and server process communicates. The logical characteristics are governed by process-to-process (or interprocess) communications protocols that give the signals meaning and purpose. It is at this level that most of the Client/Server conversation takes place.

Although the preceding analogy helps us understand the basic Client/Server interactions, it is required to have a better understanding of computer communication to better understand the flow of data and control information in a client server environment. To understand the details we will refer to Open System Interconnection (OSI) network reference model which is an effort to standardize the diverse network systems. Figure 3.7 depicts the flow of information through each layer of OSI model.

From the figure, we can trace the data flow:

- The client application generates a SQL request.
- The SQL request is sent down to the presentation layer, where it is changed to a format that the SQL server engine can understand.
- Now, the SQL request is handed down to session layer. This layer establishes the connection to the client processes with the server processes. If the database server requires user verification, the session layer generates the necessary message to log on and verify the end user. And also this layer will identify which mesaages are control messages and which are data messages.
- After the session is established and validated, the SQL request is sent to the transport layer. The transport layer generates some error validation checksums and adds some transport-layer-specific ID information.
- Once the transport layer has performed its functions, the SQL request is handed down to the network layer. This layer takes the SQL request, identifies the address of the next node in the path, divides the SQL request into several smaller packets,

and adds a sequence number to each packet to ensure that they are assembled in the correct order.

- Next the packet is handed to the data-link layer. This layer adds more control information, that depends on the network and on which physical media are used. The data-link layer sends the frame to the next node.
- When the data-link layer determines that it is safe to send a frame, it hands the frame down to the physical layer, which transmits it into a collection of ones and zeros(bits), and then transmit the bits through the network cable.
- The signals transmitted by the physical layer are received at the server end at the physical layer, which passes the data to the data-link layer. The data-link layer reconstructs the bits into frames and validates them. At this point, the data-link layer of the client and server computer may exchange additional messages to verify that the data were received correctly and that no retransmission is necessary. The packet is sent up to the network layer.
- The network layer checks the packet's destination address. If the final destination is some other node in network, the network layer identifies it and sends the packet down to the data-link layer for transmission to that node. If the destination is the current node, the network layer assembles the packets and assigns appropriate sequence numbers. Next, the network layer generates the SQL request and sends it to the transport layer.
- Most of the Client/Server "conversation" takes place in the session layer. If the communication between client and server process is broken, the session layer tries to reestablish the session. The session layer identifies and validates the request, and sends it to the presentation layer.
- The presentation layer provides additional validation and formatting.
- Finally, the SQL request is sent to the database server or application layer, where it is executed.

Although the OSI framework helps us understand network communications, it functions within a system that requires considerable infrastructure. The network protocols constitute the core of network infrastructure, because all data travelling through the network must adhere to some network protocol. In the Client/Server environment, it is not usual to work with several different network protocols. In the previous section, we noted that different server processes might support different network protocols to communicate over the network.

For example, when several processes run on the client, each process may be executing a different SQL request, or each process may access a different database server. The transport layer ID helps the transport layer identify which data corresponds to which session.

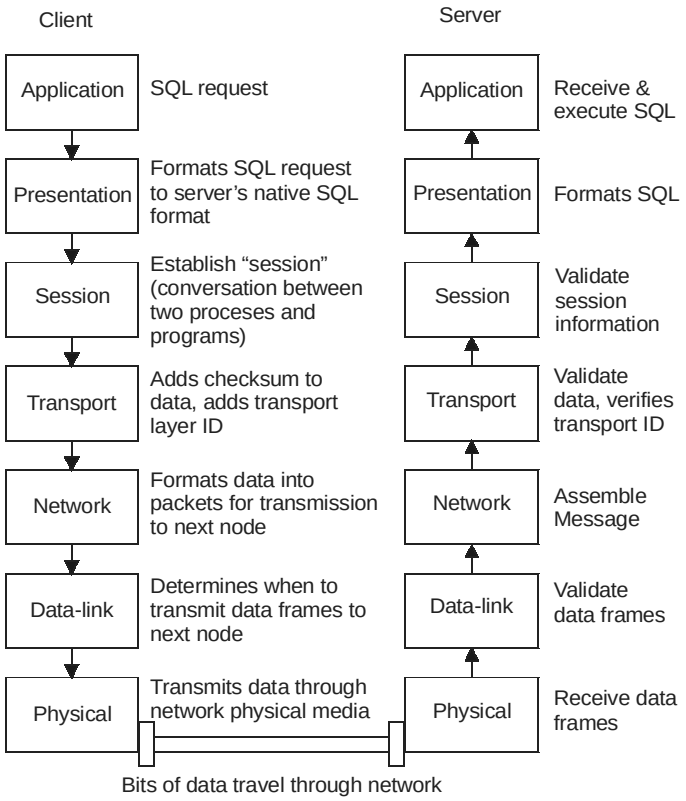


Fig.3.7: Flow of Information through the OSI Model

3.7 ARCHITECTURE FOR BUSINESS INFORMATION SYSTEM

3.7.1 Introduction

In this section, we will discuss several patterns for distributing business information systems that are structured according to a layered architecture. Each distribution pattern cuts the architecture into different client and server components. All the patterns discussed give an answer to the same question: How do I distribute a business information system? However, the consequences of applying the patterns are very different with regards to the forces influencing distributed systems design. Distribution brings a new design dimension into the architecture of information systems. It offers great opportunities for good systems design, but also complicates the development of a suitable architecture by introducing a lot of new design aspects and trap doors compared to a centralized system. While

constructing the architecture for a business information system, which will be deployed across a set of distributed processing units (e.g., machines in a network, processes on one machine, threads within one process), you are faced with the question:

How do I partition the business information system into a number of client and server components, so that my users' functional and non-functional requirements are met?

There are several answers to this question. The decision for a particular distribution style is driven by users' requirements. It significantly influences the software design and requires a very careful analysis of the functional and non-functional requirements.

3.7.2 Three-Layer Architecture

A Business Information System, in which many (spatially distributed) users work in parallel on a large amount of data. The system supports distributed business processes, which may span a single department, a whole enterprise, or even several enterprises. Generally, the system must support more than one type of data processing, such as On-Line Transaction Processing (OLTP), off-line processing or batch processing. Typically, the application architecture of the system is a *Three-Layer Architecture*, illustrated in Fig. 3.8.

The user interface handles presentational tasks and controls the dialogue the application kernel performs the domain specific business tasks and the database access layer connects the application kernel functions to a database. Our distribution view focuses on this coarse-grain component level. In developing distributed system architecture we mainly use the *Client/Server Style.* Within these model two roles, client and server classify components of a distributed system. Clients and servers communicate via a simple request/ response protocol.

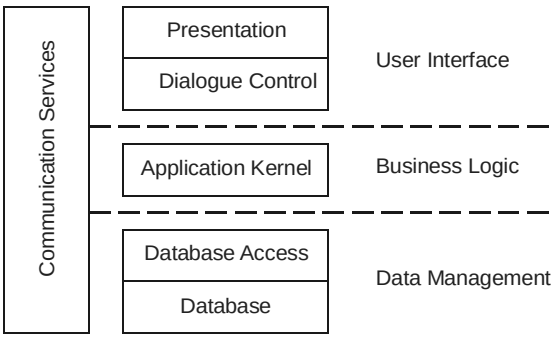


Fig.3.8: Three-Layer Architecture for Business Information System

3.7.3 General Forces

- *Business needs vs. construction complexity:* On one hand, allocating functionality and data to the places where it is actually needed supports distributed business processes

very well, but on the other hand, distribution raises a system's complexity. Client server systems tend to be far more complex than conventional host software architectures. To name just a few sources of complexity: GUI, middleware, and heterogeneous operating system environments. It is clear that it often requires a lot of compromises to reduce the complexity to a level where it can be handled properly.

- *Processing style:* Different processing styles require different distribution decisions. Batch applications need processing power close to the data. Interactive processing should be close to input/output devices. Therefore, off-line and batch processing may conflict with transaction and on-line processing.
- *Distribution vs. performance:* We gain performance by distributed processing units executing tasks in parallel, placing data close to processing, and balancing workload between several servers. But raising the level of distribution increases the communication overhead, the danger of bottlenecks in the communication network, and complicates performance analysis and capacity planning. In centralized systems the effects are much more controllable and the knowledge and experience with the involved hardware and software allows reliable statements about the reachable performance of a configuration.
- *Distribution vs. security:* The requirement for secure communications and transactions is essential to many business domains. In a distributed environment the number of possible security holes increases because of the greater number of attack points. Therefore, a distributed environment might require new security architectures, policies and mechanisms.
- *Distribution vs. consistency:* Abandoning a global state can introduce consistency problems between states of distributed components. Relying on a single, centralized database system reduces consistency problems, but legacy systems or organizational structures (off-line processing) can force us to manage distributed data sources.
- *Software distribution cost:* The partitioning of system layers into client and server processes enables distribution of the processes within the network, but the more software we distribute the higher the distribution, configuration management, and installation cost. The lowest software distribution and installation cost will occur in a centralized system. This force can even impair functionality if the software distribution problem is so big that the capacities needed exceed the capacities of your network. The most important argument for so called diskless, Internet based network computers is exactly software distribution and configuration management cost.
- *Reusability vs. performance vs. complexity:* Placing functionality on a server enforces code reuse and reduces client code size, but data must be shipped to the server and the server must enable the handling of requests by multiple clients.

3.7.4 Distribution Pattern

To distribute an information system by assigning client and server roles to the components of the layered architecture we have the choice of several distribution styles. Figure 3.9 shows the styles, which build the pattern language. To take a glance at the pattern language we give an abstract for each pattern:

- *Distributed presentation:* This pattern partitions the system within the presentation component. One part of the presentation component is packaged as a distribution unit and is processed separately from the other part of the presentation, which can be packaged together with the other application layers. This pattern allows of an easy implementation and very thin clients. Host systems with 3270-terminals is a classical example for this approach. Network computers, Internet and intranet technology are modern environments where this pattern can be applied as well.
- *Remote user interface:* Instead of distributing presentation functionality the whole user interface becomes a unit of distribution and acts as a client of the application kernel on the server side.
- *Distributed application kernel:* The pattern splits the application kernel into two parts which are processed separately. This pattern becomes very challenging if transactions span process boundaries (distributed transaction processing).

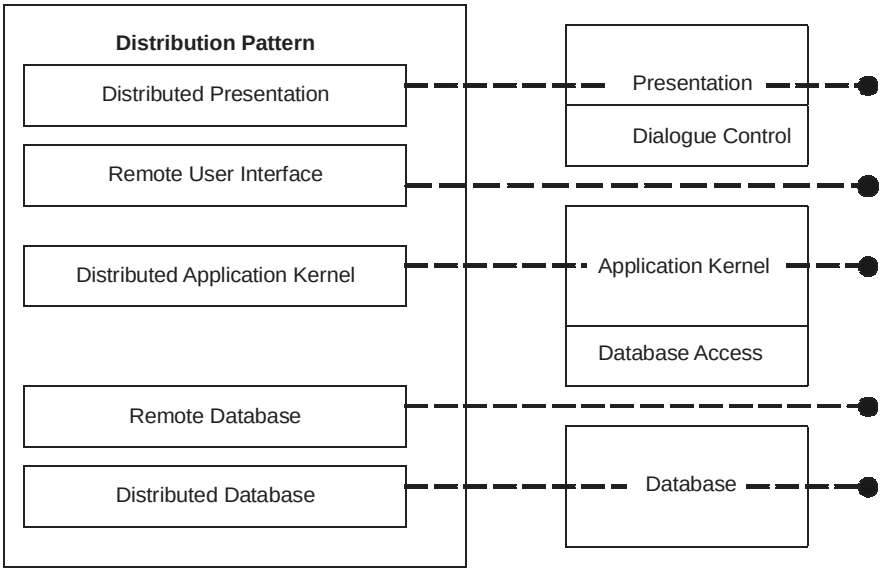


Fig.3.9: Pattern Resulting from Different Client/Server Cuts

- *Remote database:* The database is a major component of a business information system with special requirements on the execution environment. Sometimes, several applications work on the same database. This pattern locates the database component on a separate node within the system's network.

- *Distributed database:* The database is decomposed into separate database components, which interact by means of interprocess communication facilities. With a distributed database an application can integrate data from different database systems or data can be stored more closely to the location where it is processed.

3.8 EXISTING CLIENT/SERVER ARCHITECTURE

3.8.1 Mainframe-based Environment

In mainframe systems all the processing takes place on the mainframe and usually dumb terminals that are known as end user platform are used to display the data on screens.

Mainframes systems are highly centralized known to be integrated systems. Where dumb terminals do not have any autonomy. Mainframe systems have very limited data manipulation capabilities. From the application development point of view. Mainframe systems are over structured, time-consuming and create application backlogs. Various computer applications were implemented on *mainframe computers* (from IBM and others), with lots of attached (dumb, or semi-intelligent) terminals see the Fig. 3.10.

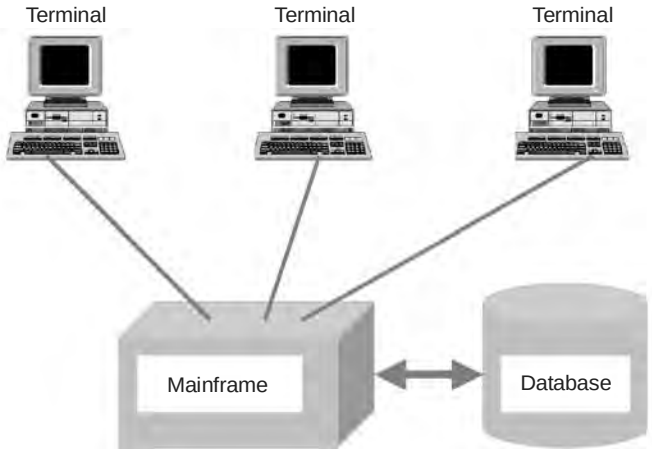


Fig.3:10: Mainframe-based Environment

There are some major problems with this approach:

- ⇨ Very inflexible.
- ⇨ Mainframe system are very inflexible.
- ⇨ Vendor lock-in was very expensive.
- ⇨ Centralized DP department was unable to keep up with the demand for new applications.

3.8.2 LAN-based Environment

LAN can be configured as a Client/Server LAN in which one or more stations called servers give services to other stations, called clients. The server version of network operating system is installed on the server or servers; the client version of the network operating system is installed on clients. A LAN may have a general server or several dedicated servers. A network may have several servers; each dedicated to a particular task for example database servers, print servers, and file servers, mail server. Each server in the Client/Server based LAN environment provides a set of shared user services to the clients. These servers enable many clients to share access to the same resources and enable the use of high performance computer systems to manage the resources.

A file server allows the client to access shared data stored on the disk connected to the file server. When a user needs data, it access the server, which then sends a copy, a print server allows different clients to share a printer. Each client can send data to be printed to the print server, which then spools and print them. In this environment, the file server station server runs a server file access program, a mail server station runs a server mail handling program, and a print server station a server print handling program, or a client print program.

Users, applications and resources are distributed in response to business requirements and linked by single Local Area Networks. See the Fig. 3.11 illustrated below:

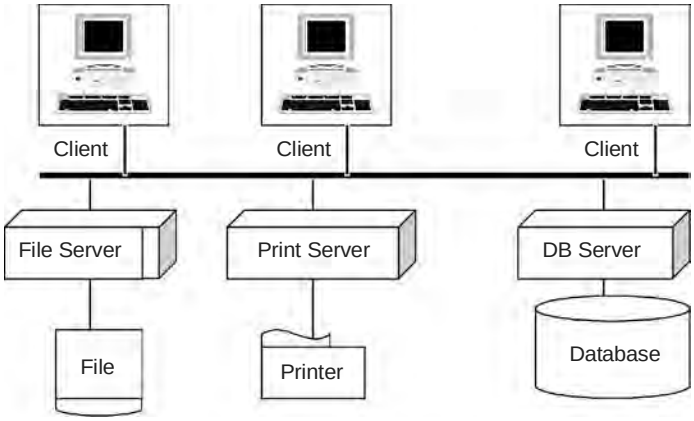


Fig.3.11: LAN Environment

3.8.3 Internet-based Environment

What the Internet brings to the table is a new platform, interface, and architectures. The Internet can employ existing Client/Server applications as true Internet applications, and integrate applications in the Web browser that would not normally work and play well together. The Internet also means that the vast amount of information becomes available

from the same application environment and the interface. That's the value. See the Fig. 3.12 given below:

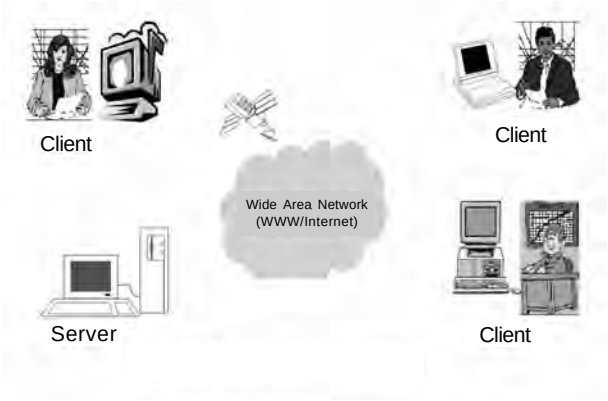


Fig.3.12: Internet-based Environment

The internet also puts fat client developers on a diet. Since most internet applications are driven from the Web server, the application processing is moving off the client and back onto the server. This means that maintenance and application deployment become much easier, and developers don't have to deal with the integration hassles of traditional Client/Server (such as loading assorted middleware and protocol stacks).

The web browsers are universal clients. A web browser is a minimalist client that interprets information it receives from a server, and displays it graphically to a user. The client is simply here to interpret the server's command and render the contents of an HTML page to the user. Web browsers-like those from Netscape and Spyglass – are primarily interpreters of HTML commands. The browser executes the HTML commands to properly display text and images on a specific GUI platform; it also navigates from one page to another using the embedded hypertext links. HTTP server produce platform independent content that clients can then request. A server does not know a PC client from a Mac client – all web clients are created equal in the eyes of their web server. Browsers are there to take care of all the platform-specific details.

At first, the Web was viewed as a method of publishing informaton in an attractive format that could be accessed from any computer on the internet. But the newest generation of the Web includes programmable clients, using such programming environments as Sun Microsystem's Java and Microsoft's ActiveX. With these programming environments, the Web has become a viable and compelling platform for developing Client/Server applications on the Internet, and also platform of choice for Client/Server computing. The World Wide Web (WWW) information system is an excellent example of client server "done right". A server system supplies multimedia documents (pages), and runs some application programs (HTML forms and CGI programs, for example) on behalf of the client. The client takes complete responsibility for displaying the hypertext document, and for the user's response to it. Whilst the majority of "real world" (i.e., commercial) applications of Client/Server are in database applications.

EXERCISE 3

1. What is the role of mainframe-centric model in Client/Server computing?
2. Explain Connectivity and Communication Interface Technology in Client/Server application. How does transmission protocol work in Client/Server application?
3. Explain Peer to Peer architecture. What is the basic difference between Client/Server and Peer to Peer Computing?
4. Draw the block diagram of Client/Server architecture and explain the advantage of Client/Server computing with the help of suitable diagram.
5. Explain shared tiered Client/Server architecture.
6. How are connectivity and interoperability between Client/Server achieved? Explain.
7. Explain Client/Server architecture. What is the basic difference between Client/Server and peer to peer computing?
8. Explain the three-level architecture of database management system. Also explain the advantages and disadvantages of DBMS.
9. Draw the block diagram of Client/Server architecture and explain the advantage of Client/Server computing with the help of suitable example.
10. What are the various ways available to improve the performance of Client/Server computing?

4

Client/Server and Databases

4.1 INTRODUCTION

Storing Data and the Database

Server translates the ink/paper storage model into an electronic/magnetic media storage model, but the fundamental arrangement is the same. The basic building block (his computer equivalent of information-on-paper) is called data. Data is information in its simplest form, meaningless until related together in some fashion so as to become meaningful. Related data is stored on server's disk under a unique name, called file. Related file are gathered together into directories, and related directories are gathered together into larger and larger directories until all the required information is stored in a hierarchy of directories on the server's hard disk.

The server's "filing cabinet" is a database; it offers a number of advantages over the paper model. A particular file can be searched electronically, even if only remembering a tiny portion of the file contains.

A database, generally defined, is a flexible, hierarchical structure for storing raw data, which facilitates its organization into useful information. All data on computer is stored in one kind of database or another. A spreadsheet is a database, storing data in an arrangement of characters and formatting instructions. What a database does, then, is breakdown information into its most fundamental components and then create meaningful relationships between those components. We depend on databases of varying configurations and complexity for all our computerized information need.

The Fig. 4.1 illustrates the evolution of database technology from the first computer in late 1950's to the object-oriented database technologies.

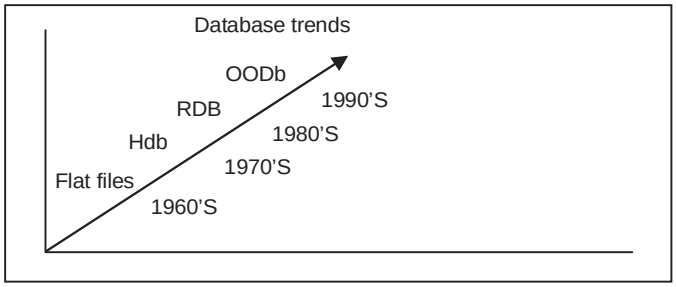


Fig.4.1: Evolution of Database Technologies

Using a database you can tag data, relating it to other data in several different ways, without having to replicate the data in different physical locations. This ability to access and organize data in a flexible manner without making physical copies of it (and thus preserving the integrity of the information at its most basic level) is what has lead to the increasing use of client/server technology as a widespread business information model.

Database System Architectures

Before proceeding to understand the Client/Server database it is quite essentials to have a brief introduction about the other available architecture of database systems.

Client/Server database system: The functionality is spilted between a server and multiple client systems, i.e., networking of computers allows some task to be executed on server system and some task to be executed on client system.

Distributed database system: Geographically or administratively distributed data spreads across multiple database systems.

Parallel database system: Parallel processing within computer system allows database system activities to be speeded up, allowing faster response to transaction; queries can be preceded in a way that exploits the parallelism offered by the underlying computer system.

Centralized database system: Centralized database systems are those run on a single system and do not interact with other computer systems. They are single user database systems (on a PC) and high performance database system (on high end server system).

4.2 CLIENT/SERVER IN RESPECT OF DATABASES

4.2.1 Client/Server Databases

Servers exist primarily to manage databases of information in various formats. Without the database, servers would be impractical as business tools. True, you could still use them to share resources and facilitate communication; but, in the absence of business database, a peer-to-peer network would be a more cost effective tool to handle these jobs. So the question of client server becomes a question of whether or not your business needs centralized database. Sharing and communications are built on top of that.

A Database Management System (DBMS) lies at the center of most Client/Server systems in use today. To function properly, the Client/Server DBMS must be able to:

- Provide transparent data access to multiple and heterogeneous clients, regardless of the hardware, software, and network platform used by the client application.
- Allow client request to the database server (using SQL requests) over the network.
- Process client data requests at the local server.
- Send only the SQL result to the clients over the network.

A Client/Server DBMS reduces network traffic because only the rows that match the query are returned. Therefore, the client computer resources are available to perform other system chores such as the management of the graphical user interface. Client/Server DBMS differ from the other DBMSs in term of where the processing take place and what data are sent over the network to the client computer. However, Client/Server DBMSs do not necessarily require distributed data.

Client/Server systems changes the way in which we approach data processing. Data may be stored in one site or in multiple sites. When the data are stored in multiple sites, Client/Server databases are closely related to distributed databases.

4.2.2 Client/Server Database Computing

Client/Server database computing evolved in response to the drawbacks of the mainframe (very expensive operating cost because they require specialized operational facilities demand expensive support, and do not use common computer components), and PC/file server computing environments (the drawback of PC-based computing is that all RDBMS processing is done on the local PC, when a query is made to the file server, the file server does not process the query, instead, it returns the data required to process the query, this can result in decreased performance and increased network bottlenecks). By combining the processing power of the mainframe and the flexibility and price of the PC, Client/Server database computing encompasses the best of both words.

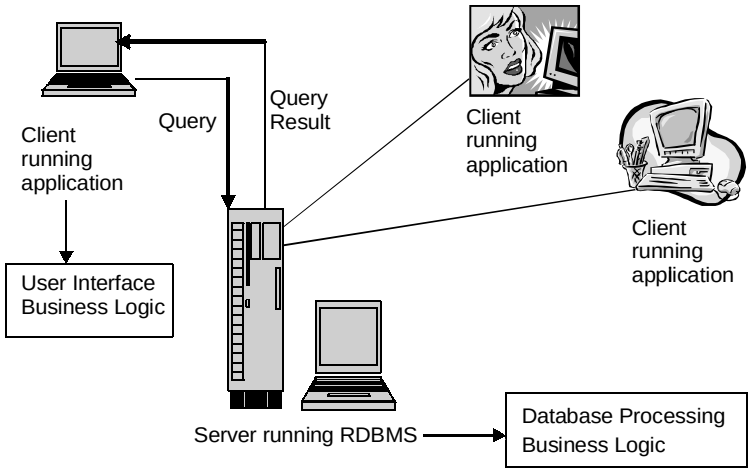


Fig. 4.2: Client/Server Database Computing

Client/Server database computing can be defined as the logical partition of the user interface, database management, and business; logic between the client computer and server computer. The network links each of these processes. The client computer, also called workstation, controls the user interface. The client is where text and images are displayed to the user and where the user inputs data. The user interface can be text based or graphical based. The server computer controls database management. The server is where data is stored, manipulated, and stored. In the Client/Server database environment, all database processing occurs on the server.

Business logic can be located on the server, on the client, or mixed between the two. This type of logic governs the processing of the application.

Client/Server database computing vs. Mainframe and PC/file server computing

Client/Server database computing is preferred in comparison to other database computing. Following are the reasons for its popularity:

Affordability: *Client/Server computing can be less expensive than mainframe computing. The underlying reason is simple: Client/Server computing is based on an open architecture, which allows more vendors to produce competing products, which drives the cost down. This is unlike mainframe-based systems, which typically use proprietary components available only through a single vendor. Also, Client/Server workstations and servers are often PC based. PC prices have fallen dramatically over the years, which has led to reduce Client/Server computing costs.*

Speed: *The separation of processing between the client and the server reduces the network bottlenecks, and allows a Client/Server database system to deliver mainframe performance while exceeding PC/file server performance.*

Adaptability: *The Client/Server database computing architecture is more open than the proprietary mainframe architecture. Therefore, it is possible to build an application by selecting an RDBMS from one vendor, hardware from another vendor. Customers can select components that best fit their needs.*

Simplified data access: *Client/Server database computing makes data available to the masses. Mainframe computing was notorious for tracking huge amounts of data that could be accessed only by developers. With Client/Server database computing, data access is not limited to those who understand procedural programming languages (which are difficult to learn and require specialized data access knowledge). Instead, data access is providing by common software products tools that hide the complexities of data access. Word processing, spreadsheet, and reporting software are just a few of the common packages that provide simplified access to Client/Server data.*

4.3 CLIENT/SERVER DATABASE ARCHITECTURE

Relational database are mostly used by Client/Server application, where the server is a database server. Interaction between client and server is in the form of transaction in which client makes database request and receives a database response.

In the architecture of such a system, server is responsible for maintaining the database, for that purpose a complex database management system software module is required.

Various types of applications that make use of the database can install on client machine. The "glue" that ties client and server together is software that enables the client to make request for access to the server's database, that is SQL (Structured Query Language), shown in the Fig. 4.3 given below:

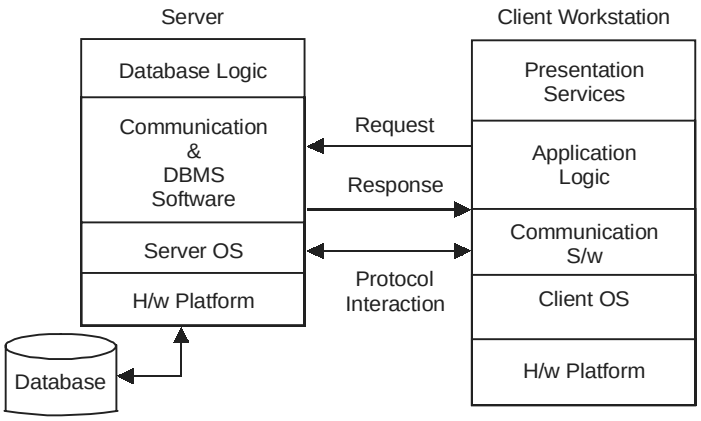


Fig.4.3: Client/Server Database Architecture

According to this architecture, all the application logic (software used for data analysis) is residing on the client side, while the server is concerned with managing the database. Importance of such architecture depends on the nature of application, where it is going to be implemented. And the main purpose is to provide on line access for record keeping. Suppose a database with million of records residing on the server, server is maintaining it. Some user wants to fetch a query that result few records only. Then it can be achieved by number of search criteria. An initial client query may yield a server response that satisfies the search criteria. The user then adds additional qualifiers and issues a new query. Returned records are once again filtered. Finally, client composes next request with additional qualifiers. The resulting search criteria yield desired match, and the record is returned to the client. Such Client/Server architecture is well-suited for such types of applications due to:

- Searching and sorting of large databases are a massive job; it requires large disk space, high speed CPU along with high speed Input/Output architecture. On the other hand, in case of single user workstations such a storage space and high power is not required and also it will be costlier.
- Tremendous traffic burden is placed on the network in order to move the million of records to the clients for searching, then it is not enough for the server to just be able to retrieve records on behalf of a client; the server needs to have database logic that enables it to perform searches on behalf of a client.

Various types of available Client/Server Database Architecture are discussed in detail; in the section given:

(*i*) Process-per-client architecture.

(*ii*) Multi-threaded architecture.

(*iii*) Hybrid architecture.

(*i*) Process-per-client architecture: As the name reveals itself server process considers each client as a separate process and provides separate address space for each user. Each process can be assigned to a separate CPU on a SMP machine, or can assign processes to a pool of available CPUs. As a result, consumes more memory and CPU resources than other schemes and slower because of process context switches and IPC overhead but the use of a TP Monitor can overcome these disadvantages. Performance of Process-per-client architecture is very poorly when large numbers of users are connecting to a database server. But the architecture provides the best protection of databases.

Examples of such architecture is DB2, Informix, and Oracle6, Fig. 4.4 illustrates such architecture.

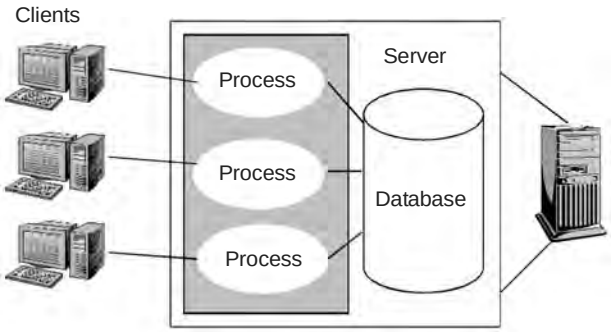


Fig.4.4: Process-per-client Architecture

(*ii*) Multi-threaded architecture: Architecture supports a large numbers of clients running a short transaction on the server database. Provide the best performance by running all user requests in a single address space. But do not perform well with large queries. Multi-threaded architecture conserves Memory and CPU cycles by avoiding frequent context switches. There are more chances of portability across the platforms.

But it suffers by some drawback first in case of any misbehaved user request can bring down the entire process, affecting all users and their requests and second long-duration tasks of user can hog resources, causing delays for other users. And the architecture is not as good at protection point of view. Some of the examples of such architecture are: Sybase, Microsoft SQL Server, and illustrated in Fig. 4.5.

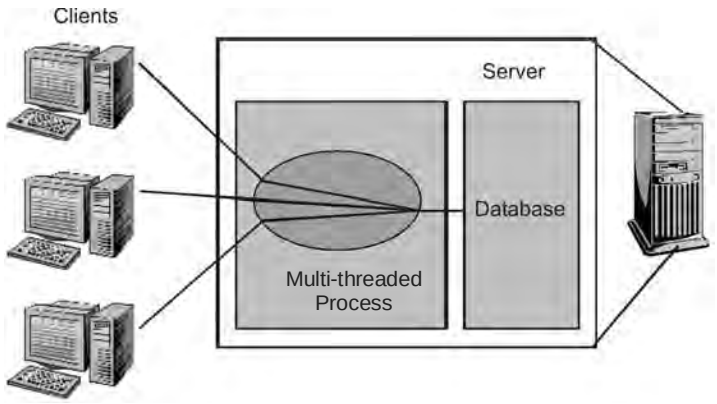


Fig.4.5: Multi-threaded Architecture

(*iii*) Hybrid architecture: Hybrid architecture provides a protected environment for running user requests without assigning a permanent process for each user. Also provides the best balance between server and clients. Hybrid architecture of Client/Server Database is basically comprised of three components:

- Multi-threaded network listener: Main task of this is to assign client connection to a dispatcher.
- Dispatcher processes: These processes are responsible for placing the messages on an internal message queue. And finally, send it back to client when response returned from the database.
- Reusable, shared, worker processes: Responsible for picking work off the message queue and execute that work and finally places the response on an output message queue.

Hybrid architecture of Client/Server database suffer from queue latencies, which have an adversely affect on other users. The hybrid architecture of Client/Server database is shown in Fig. 4.6 given below. Some of the examples of such architectures are Oracle7i and Oracle8i/9i.

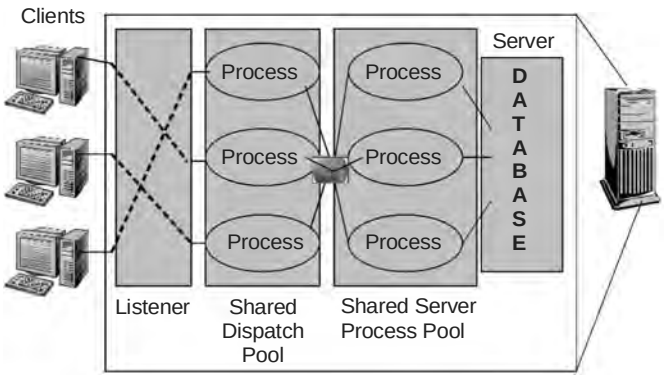


Fig.4.6: Hybrid Architecture

4.4 DATABASE MIDDLEWARE COMPONENT

As we have already discussed in Client/Server architecture that communication middleware software provides the means through which clients and servers communicate to perform specific actions. This middleware software is divided into three main components. As shown in the Fig. 4.7 given below:

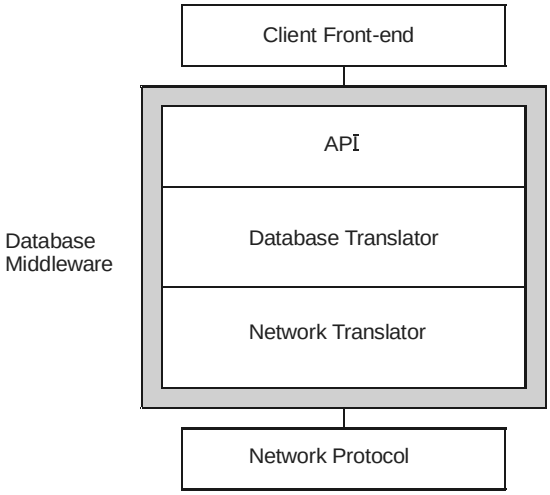


Fig.4.7: Database Middleware Components

- Application programming interface.
- Database translator.
- Network translator.

These components (or their functions) are generally distributed among several software layers that are interchangeable in a plug and play fashion.

The *application-programming interface* is public to the client application. The programmer interacts with the middleware through the APIs provided by middleware software. The middleware API allows the programmer to write generic SQL code instead of code specific to each database server. In other words, the middleware API allows the client process to be database independent. Such independence means that the server can be changed without requiring that the client applications be completely rewritten.

The *database translator* translates the SQL requests into the specific database server syntax. The database translator layer takes the generic SQL request and maps it to the database server's SQL protocol. Because a database server might have some non-standard features, the database translator layer may opt to translate the generic SQL request into the specific format used by the database server.

If the SQL request uses data from two different database servers, the database translator layer will take care of communicating with each server, retrieving the data using the common format expected by the client application.

The network translator manages the network communication protocols. Remember that database server can use any of the network protocols. If a client application taps into the two databases, one that uses TCP/IP and another that uses IPX/SPX, the network layer handles all the communications detail of each database transparently to the client application. Figure 4.8 illustrates the interaction between client and middleware database components.

Existence of these three middleware components reveals some benefits of using middleware software; according to that clients can:

- Access multiple (and quite different) databases
- Be database-server-independent
- Be network-protocol-independent

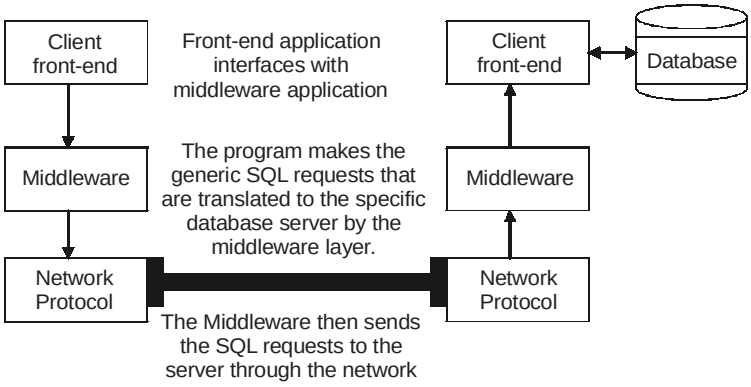


Fig.4.8: Interaction Between Client/Server Middleware Components

4.5 ACCESS TO MULTIPLE DATABASES

To understand how the three components of middleware database work together, lets us see how a client accesses two different databases servers. The Fig. 4.9 shows a client application request data from an oracle database server (Oracle Corporation) and a SQL Server database server (Microsoft Corporation). The Oracle database server uses SQL *Net as its communications protocol with the client; the SQL Server uses Named Pipes as the communications protocol. SQL *Net, a proprietary solution limited to Oracle database, is used by Oracle to send SQL request over a network. Named Pipes is a inter-process communication (IPC) protocol common to multitasking operating systems such as UNIX and OS/2, and it is used in SQL Server to manage both client and server communications across the network.

As per the Fig. 4.9, it is notable that the Oracle server runs under the UNIX operating system and uses TCP/IP as its network protocol. The SQL Server runs under the Windows NT operating system and uses NetBIOS as its network protocol. In this case, the client application uses a generic SQL query to access data in two tables: an Oracle table and a SQL Server table. The database translator layer of middleware software contains two modules, one for each database server type to be accessed.

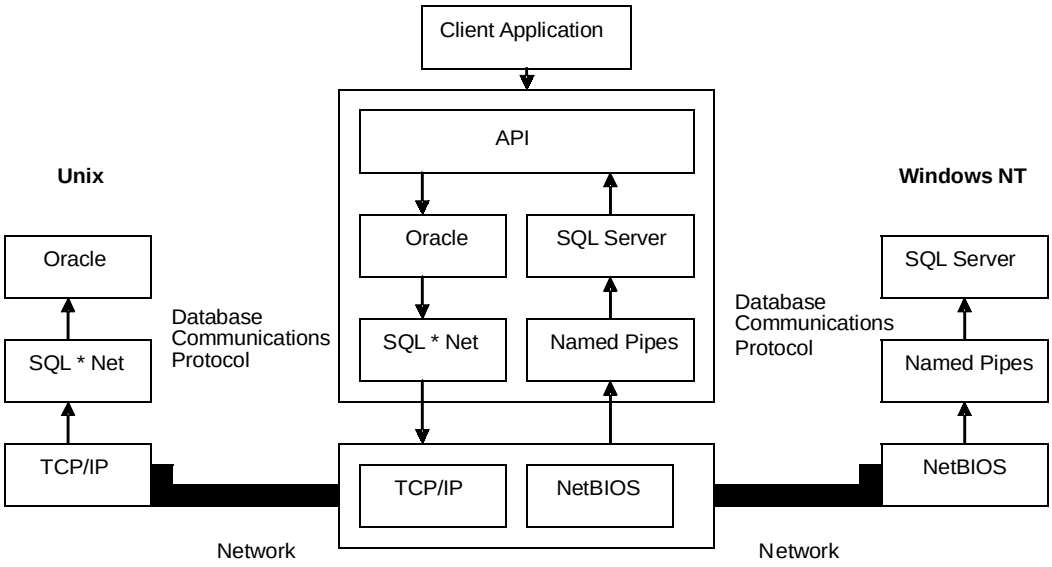


Fig.4.9: Multiple Database Server Access Through Middleware

Each module handles the details of each database communications protocol. The network translator layer takes care of using the correct network protocol to access each database. When the data from the query are returned, they are presented in a format common to the client application. The end user or programmer need not be aware of the details of data retrieval from the servers. Actually, the end user might not even know where the data reside or from what type of DBMS the data were retrieved.

4.6 DISTRIBUTED CLIENT/SERVER DATABASE SYSTEMS

Data Distribution

Distributed data and *distributed processing* are terms used widely in the word of Client/ Server computing. The differences in these two can be easily understood by the two figures 4.10(a) and 4.10(b). Distributed data refers to the basic data stored in the server, which is distributed to different members of the work team. While distributed processing refers to the way different tasks are organized among members of the work team. If a set of information handling tasks is thought of as a single step-by-step process and is split among

the members of the work-team so that they can handle the steps more efficiently, that process is distributed.

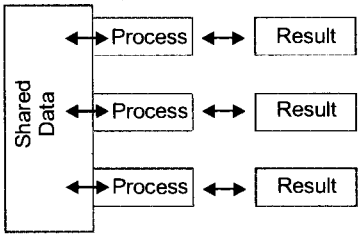


Fig.4.10(*a*): Distributed Data

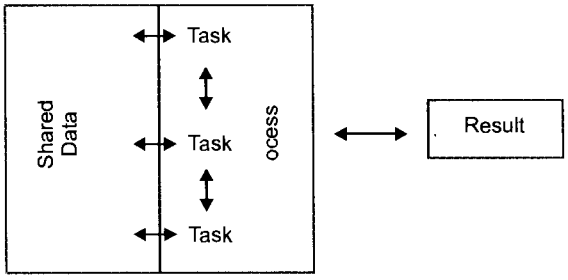


Fig.4.10(*b*): Distributed Processing

Here's an example: Imagine that a customer's ordering and payment information is stored in a central customer record on the server. This record is accessed by many departments in various locations throughout the company (accounting and shipping/ receiving, to name two). Thus the data is an example of distributed data. In addition, because accounting and shipping/receiving work with the data in unique but related ways in order to accomplish a specific goal (updating the customer record), their activities are an example of distributed processing.

Distributed Client/Server database system must have some characteristics that are discussed in this section. The location of data is transparent to the user. The data can be located in the local PC, the department server, or in a mainframe across the country. The data can also be distributed among different locations and among different databases using the same or even the different data models.

The data in database can be partitioned in several ways. And the partitioned data may be allocated (process of data allocation describes where to locate the data, some data allocation strategies are: Centralized, partitioned, replicated) in several different ways, and the data may be replicated in several nodes see the Fig. 4.11.

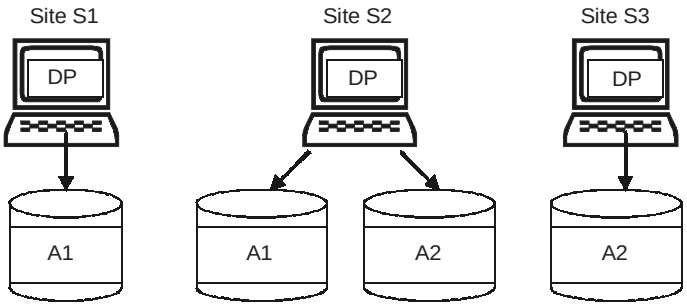


Fig.4.11: Data Replication

For example, suppose database A is divided into two fragments A1 and A2. Within a replicated distributed database, the scenario depicted in figure 4.11 is possible: fragment A1 is stored at sites S1 and S2, while fragment A2 is stored at sites S2 and S3. The network can be a LAN, a MAN, or a WAN. The user does not need to know the data location, how to get there, or what protocols are used to get there.

- Data can be accessed and manipulated by the end user at any time in many ways. Data accessibility increases because end users are able to access data directly and easily, usually by pointing and clicking in their GUI-based system. End user can manipulate data in several ways, depending on their information needs. For example, one user may want to have a report generated in a certain format, whereas another user may prefer to use graphical presentations. Powerful applications stored on the end user's side allow access and manipulation of data in a way that were never before available. The data request is processed on the server side; the data formatting and presentation are done on the client side.
- The processing of data (retrieval, storage, validation, formatting, presentation and so on) is distributed among multiple computers. For example, suppose that a distributed Client/Server system is used to access data from three DBMSs located at different sites. If a user requests a report, the client front-end will issue a SQL request to the DBMS server. The database server will take care of locating the data; retrieving it from the different locations, assembling it, and sending it back to the client. In this scenario, the processing of the data access and retrieval.

4.7 DISTRIBUTED DBMS

Client/Server database is commonly known for having distributed database capabilities. But is not necessarily able to fulfil the entire required Client/Server characteristics that are in need for particular system. Client/Server architecture refers to the way in which computers interact to form a system. The Client/Server architecture features a user of resources, or a client, and a provider of resources, or a server. The Client/Server architecture

can be used to implement a DBMS in which the client is Transaction Processor and the server is the Data Processor. Client/Server interaction in a DDBMS are carefully scripted. The client (TP) interacts with the end use and sends a request to the server (DP). The server receives, schedules, and executes the request, selecting only those records that are needed by the client. The server then sends the data to the client only when the client requests the data. The database management system must be able to manage the distribution of data among multiple nodes. The DBMS must provide distributed database transparency features like:

- Distribution transparency.
- Transaction transparency.
- Failure transparency.
- Performance transparency.
- Heterogeneity transparency.

Number of relational DBMS, which are started as a centralized system with its components like user interface and application programs were moved to the client side. With standard language SQL, creates a logical dividing point between client and server. Hence, the query and transaction functionality remained at the server side. When DBMS access is required, the program establishes a connection to the DBMS; which is on the server side and once the connection is created, the client program can communicate with the DBMS.

Exactly how to divide the DBMS functionality between client and server has not yet been established. Different approaches have been proposed. One possibility is to include functionality of a centralized DBMS at the server level. A number of relational DBMS concepts have taken this approach, where an SQL server is provided to the clients. Each client must then formulate the appropriate SQL queries and provide the user interface and programming language interface functions. Since SQL is a relational standard, various SQL servers possibly provided by different vendors, can accept SQL commands. The client may also refer to a data dictionary that includes information on the distribution of data among the various SQL servers, as well as modules for decomposing a global query into a number of local queries that can be executed at the various sites. Interaction between client and server might proceed as follows during the processing of an SQL query:

- The client passes a user query and decomposes it into a number of independent site queries. Each site query is sent to the appropriate server site.
- Each server process the local query and sends the resulting relation to the client site.
- The client site combines the results of the subqueries to produce the result of the originally submitted query.

In this approach, the SQL server has also been called a transaction server or a Database Processor (DP) or a back-end machine, whereas the client has been called Application

Processor (AP) or a front-end machine. The interaction between client and server can be specified by the user at the client level or via a specialized DBMS client module that is part of DBMS package. For example, the user may know what data is stored in each server, break-down a query request into site subqueries manually, and submit individual subqueries to the various sites. The resulting tables may be combined explicitly by a further user query at the client level. The alternative is to have the client module undertake these actions automatically.

In a typical DBMS, it is customary to divide the software module into three levels:

L1: The server software is responsible for local data management at site, much like centralized DBMS software.

L2: The client software is responsible for most of the distributions; it access data distribution information from the DBMS catalog and process all request that requires access to more than one site. It also handles all user interfaces.

L3: The communication software (sometimes in conjunction with a distributed operating system) provides the communication primitives that are used by the client to transmit commands and data among the various sites as needed. This is not strictly part of the DBMS, but it provides essential communication primitives and services.

The client is responsible for generating a distributed execution plan for a multisite query or transaction and for supervising distributed execution by sending commands to servers. These commands include local queries and transaction to be executed, as well as commands to transmit data to other clients or servers. Hence, client software should be included at any site where multisite queries are submitted. Another function controlled by the client (or coordinator) is that of ensuring consistency of replicated copies of a data item by employing distributed (or global) concurrency control techniques. The client must also ensure the atomicity of global transaction by performing global recovery when certain sites fail. One possible function of the client is to hide the details of data distribution from the user; that is, it enables the user to write global queries and transactions as through the database were centralized, without having to specify the sites at which the data references in the query or transaction resides. This property is called distributed transparency. Some DDBMSs do not provide distribution transparency, instead requiring that users beware of the details of data distribution. In fact, there is some resemblance in between Client/Server and DDBMS. The Client/Server system distributes data processing among several sites, whether as the DDBMS distributes the data at different locations, involving some complimentary and overlapping functions. DDBMS use distributed processing to access data at multiple sites.

4.8 WEB/DATABASE SYSTEM FOR CLIENT/SERVER APPLICATIONS

Nowadays, almost all the MNC's providing information and performing all their activities online through internet or intranet. In this way, the information retrieval becomes quick

and easier. It is obvious that all kinds of information the corporate world is providing through web pages. Also through links on home page they provides the facilities to enter into the corporate intranet, whether it is finance, human resource, sales, manufacturing or the marketing department. Departmental information as well as services can be accessed from web pages Even the web is powerful and flexible tool for supporting corporate requirements but they provides a limited capability for maintaining a large, change base of data. To get effectiveness on Intranet/Internet the organizations are connecting the web services to a database with its own database management systems.

Web-database integration has been illustrated in Fig. 4.12 shown below; a client machine that runs a web browser issues a request for information in the form of a URL (Uniform Resource Locator) reference. This reference triggers a program at the web server that issues the correct database command to a database server. The output returned to the web server is converted into a HTML format and returned to the web browser.

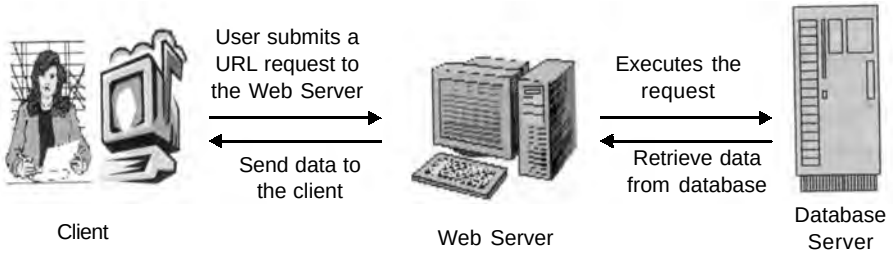


Fig.4.12: Web Database System Integration

4.8.1 Web/Database vs. Traditional Database

The section given below lists the advantages of a web/database system compared to a more traditional database approach.

- *Administration:* The only connection to the database server is the web server. The addition of a new type of database server does not require configuration of all the requisite drivers and interfaces at each type of client machine. Instead, it is only necessary for the web server to be able to convert between HTML and the database interface.
- *Deployment:* Browsers are already available across almost all platforms. Which relieves the developer of the need to implement graphical user interface across multiple customer machines and operating systems? In addition, developers can assume that customers already have and will be able to use browsers as soon as the internet web server is available. Avoiding deployment issues such as installation and synchronized activation.

- *Speed:* Large portion of the normal development cycle, such as development and client design, do not apply to web-based projects. In addition, the text based tags of HTML allow for rapid modification, making it easy to continually improve the look and feel of the application based on the ser feedback. By contrast, changing form or content of a typical graphical-based application can be a substantial task.
- *Information presentation:* Hypermedia base of the web enables the application developers to employ whatever information structure is best for given application, including the use of hierarchical formats in which progressive levels of detail are available to the user.

The section follows lists the disadvantages of a web/database system compared to a more traditional database approach.

- *Functionality*: Compared to the functionality available with a sophisticated graphical ser interface, a typical web browser interface is limited.
- *Operations*: The nature of the HTTP is such that each interaction between a browser and a server is a separate transaction, independent of prior or future exchanges. Typically, the web server keeps no information between transactions to track the states of the user.

EXERCISE 4

1. Explain the DBMS concept in Client/Server architecture in brief.
2. Is the structure of the data important to consider for processing environments? Discuss.
3. If the two servers process the same database, can it be called a Client/Server system? Explain with example.
4. One disadvantage of Client/Server system concerns control in a Database Management environment – explain the disadvantages with an example.
5. "Resource sharing architecture is not suitable for transaction processing in Client/Server environment." Discuss.
6. Compare the object-oriented and relational database management system.
7. Discuss some types of database utilities, tools and their functions.
8. What are the responsibilities of the DBA and the database designers? Also discuss the capabilities that should be provided by a DBMS.

5

Client/Server Application Components

5.1 INTRODUCTION

A Client/Server application stand at a new threshold brought on by the exponential increase of low cost bandwidth on Wide Area Networks, for example, the Internet and CompuServe; and shows a new generation of network enabled, multi-threaded desktop operating systems, for example, OS/2 Warp Connect and Windows 95. This new threshold marks the beginning of a transition from Ethernet Client/Server to intergalactic Client/Server that will result in the irrelevance of proximity. The center of gravity is shifting from single server, two tiers; LAN based departmental Client/Server to a post scarcity form of Client/Server where every machine on the global information highway can be both a client and a server. When it comes to intergalactic Client/Server applications, the imagination is at the controls. The promise of high bandwidth at very low cost has conjured visions of an information highway that turns into the world's largest shopping mall. The predominant vision is that of an electronic bazaar of planetary proportions replete with boutiques, department stores, bookstores, brokerage services, banks, and travel agencies. Like a Club Med, the mall will issue its own electronic currency to facilitate round the clock shopping and business to business transactions. Electronic agents of all kinds will be roaming around the network looking for bargains and conducting negotiations with other agents. Billions of electronic business transactions will be generated on a daily basis. Massive amounts of multimedia data will also be generated, moved, and stored on the network.

5.2 TECHNOLOGIES FOR CLIENT/SERVER APPLICATION

Some key technologies are needed at the Client/Server application level to make all this happen, including:

- **Rich transaction processing:** In addition to supporting the venerable flat transaction, the new environment requires nested transactions that can span across multiple servers, long-lived transactions that execute over long periods of time as they travel from server to server, and queued transactions that can be used in secure business-to-business dealings. Most nodes on the network should be able to participate in a secured transaction; super server nodes will handle the massive transaction loads.
- **Roaming agents:** The new environment will be populated with electronic agents of all types. Consumers will have personal agents that look after their interests; businesses will deploy agents to sell their wares on the network; and sniffer agents will be sitting on the network, at all times, collecting information to do system management or simply looking for trends. Agent technology includes cross-platform scripting engines, workflow, and Java-like mobile code environments that allow agents to live on any machine on the network.
- **Rich data management:** This includes active multimedia compound documents that you can move, store, view, and edit in-place anywhere on the network. Again, most nodes on the network should provide compound document technology — for example, OLE or OpenDoc – for doing mobile document management. Of course, this environment must also be able to support existing record-based structured data including SQL databases.
- **Intelligent self-managing entities:** With the introduction of new multi-threaded, high-volume, network-ready desktop operating systems; we anticipate a world where millions of machines can be both clients and servers. However, we can't afford to ship a system administrator with every $99 operating system. To avoid doing this, we need distributed software that knows how to manage and configure itself and protect itself against threats.
- **Intelligent middleware:** The distributed environment must provide the semblance of a single-system-image across potentially millions of hybrid Client/Server machines. The middleware must create this Houdini-sized illusion by making all servers on the global network appear to behave like a single computer system. Users and programs should be able to dynamically join and leave the network, and then discover each other. You should be able to use the same naming conventions to locate any resource on the network.

5.3 SERVICE OF A CLIENT/SERVER APPLICATION

In this section, the discussion is about most widely used five types of Client/Server applications. In no way is this meant to cover all Client/Server applications available today. The truth, there is no agreement within the computer industry as to what constitutes Client/Server and therefore, what one expect or vendor may claim to be Client/Server may not necessarily fit the definition of others.

In general, Client/Server is a system. It is not just hardware or software. It is not necessarily a program that comes in a box to be installed onto your computer's hard drive (although many software manufacturers are seeing the potential market for Client/Server products, and therefore are anxious to develop and sell such programs). Client/Server is a conglomeration of computer equipment, infrastructure, and software programs working together to accomplish computing tasks which enable their users to be more efficient and productive. Client/Server applications can be distinguished by the nature of the service or type of solutions they provide. Among them five common types of solutions are as given below.

- File sharing.
- Database centered systems.
- Groupware.
- Transactional processing.
- Distributed objects.

- **File sharing:** File sharing is Client/Server in its most primitive form. It is the earliest form of computing over a network. Some purists would deny that file sharing is Client/Server technology. In file sharing, a client computer simply sends a request for a file or records to a file server. The server, in turn, searches its database and fills the request. Usually, in a file sharing environment, the users of the information have little need for control over the data or rarely have to make modifications to the files or records. File sharing is ideal for organizations that have shared repositories of documents, images, large data objects, read-only files, etc.
- **Database centered systems:** The most common use of Client/Server technology is to provide access to a commonly shared database to users (clients) on a network. This differs from simple file sharing in that a database centered system not only allows clients to request data and data-related services, but it also enables them to modify the information on file in the database. In such systems, the database server not only houses the database itself; it helps to manage the data by providing secured access and access by multiple users at the same time. Database-centered systems utilize SQL, a simple computer language which enables data request and fulfillment messages to be understood by both clients and servers. Database-centered Client/Server applications generally fall into one of two categories:

  (*i*) Decision-Support Systems (DSS) or

  (*ii*) Online Transaction Processing (OLTP).

Both provide data on request but differ in the kinds of information needs they fulfill.

(*i*) Decision-support systems (DSS): Decision-Support Systems (DSS) are used when clients on the system frequently do analysis of the data or use the data to create reports and other documents. DSS provides a "snapshot" of data at a

particular point in time. Typically, DSS might be utilized for a library catalog, WWW pages, or patient records in a doctor's office.

(*ii*) Online transaction processing: Online Transaction Processing (OLTP) provides current, up-to-the-minute information reflecting changes and continuous updates. Users of an OLTP system typically require mission-critical applications that perform data access functions and other transactions with a one to two seconds response time.

Airline reservations systems, point-of-sale tracking systems (i.e., "cash registers" in large department stores or super markets), and a stockbroker's workstation are OLTP applications.

Structured Query Language (SQL): SQL, pronounced "sequel", stands for Structured Query Language. It is a simple set of commands which allows users to control sets of data. Originally developed by IBM, it is now the predominant database language of mainframes, minicomputers, and LAN servers. It tells the server what data the client is looking for, retrieves it, and then figures out how to get it back to the client.

SQL has become the industry standard for creating shared databases having received the "stamp of approval" from the ISO and ANSI. That's important since prior to this, there was no one commonly agreed upon way to create Client/Server database applications. With standardization, SQL has become more universal which makes it easier to set up Client/Server database systems in multi-platform/multi-NOS environments.

* **Groupware**

Groupware brings together five basic technologies multimedia document management, workflow, scheduling, conferencing, and electronic mail, in order to facilitate work activities. One author defines groupware as "software that supports creation, flow, and tracking of non-structured information in direct support of collaborative group activity." Groupware removes control over documents from the server and distributes it over a network, thus enabling collaboration on specific tasks and projects. The collaborative activity is virtually concurrent meaning that clients on the network, wherever they may be, can contribute, produce, and modify documents, and in the end, using the management and tracking features, synchronizes everything and produces a collective group product.

Multimedia Document Managements (MMDM)

With groupware, clients can have access to documents and images as needed. Multimedia document management (MMDM) allows them to take those documents and modify them. The modifications can take place in real time with several clients making changes and modifications simultaneously, or they can be modified, stored on the server for review or future action by other clients. MMDM is, in essence, an electronic filing cabinet that holds documents in the form of text, images, graphics, voice clips, video, and other media.

Workflow

Workflow refers to technologies which automatically route events (work) from one program to the next in a Client/Server groupware environment. It determines what needs to be done to complete a task or project, then merges, transforms, and routes the work item through the collaborative process.

Workflow is especially applicable to routine tasks such as processing insurance claims or income tax return preparation.

Scheduling (or Calendaring)

Scheduling or calendaring is native to groupware technology. It electronically schedules things like meetings and creates shared calendars and "to do" lists for all users on client workstations. It can work with the workflow function to monitor progress on a project (i.e., monitoring the project completion timeline), schedule a meeting or conference among key persons if needed, and notify them by automatic e-mail.

Conferencing

Conferencing is another native groupware application. This allows users at different client workstations to hold "electronic meetings." These meetings can be either "real time" or "anytime." In real time conferencing clients are interacting simultaneously. With "anytime" conferencing, documents and messages are posted to bulletin boards and clients can add their "two cents" in the form of comments, messages, or modifications.

Electronic Mail (E-mail)

E-mail is an essential element of groupware technology. It facilitates communication among clients on a network. In a groupware environment, the e-mail can be integrated with the multimedia document management, workflow, scheduling/calendaring, and conferencing features.

- **Transactional Processing**

To create truly effective Client/Server solutions, the various components within the system (the application software, the network operating system, utilities, and other programs) need to work together in unison. If infrastructure and software which enable Client/Server computing are musicians in a symphony, transaction processing would be the conductor.

- **Distributed Objects**

A distributed object is a vague term used to describe the technologies which allow clients and servers from different technologies from different environments and platforms to work seamlessly together. Its goal is to provide users with single-image, easy-to-use, virtually transparent applications.

Distributed object technology is still in its infancy and has yet to fulfill its promise of making Client/Server into the flexible, robust, intelligent, and self-managing systems that most users want and expect. Distributed objects technology has great "potential" but at this point in time, it remains just that potential.

5.4 CATEGORIES OF CLIENT/SERVER APPLICATIONS

There are variety of ways to divide the processing between client and server. But the exact distribution of data and application programming depends on the nature of the database, the type of application supported, the availability of interoperable vendor equipment, and the usage patterns within an organization. Depending on the database applications various classes of Client/Server Application has been characterized.

(*i*) Host-based processing.
(*ii*) Server-based processing.
(*iii*) Client-based processing.
(*iv*) Cooperative processing.

(*i*) Host-based processing: Virtually all the processing is done on a central host, often user interface is via a dumb terminal. It is mainly mainframe environment, not true Client/Server computing. In such a processing's workstations have very limited role as shown in Fig. 5.1 given below:

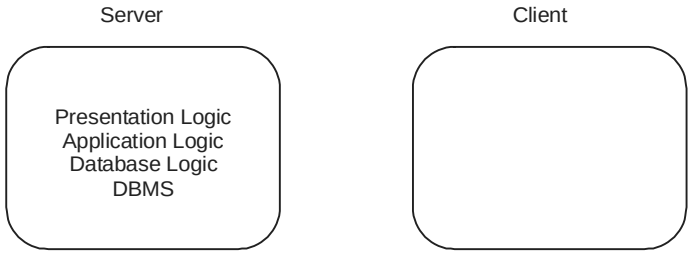


Fig.5.1: Host-base Processing

(*ii*) Server-based processing: All the processing is done on the server, and server is responsible for providing graphical user interface. A considerable fraction of the load is on the server, so this is also called *fat server* model shown in Fig. 5.2 given below:

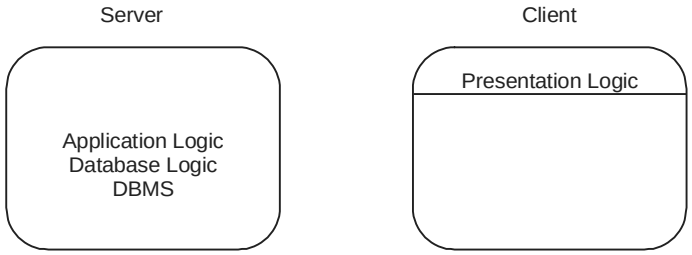


Fig.5.2: Server-base Processing

(*iii*) Client-based processing: Virtually all application processing may be done at the client, with the exception of data validation routines and other database logic functions that are best performed at the server. Some of the sophisticated database logic functions residing on the client side. This architecture is the most common Client/Server approach in current use. It enables the user to employ applications tailored to local need shown in Fig 5.3 given below:

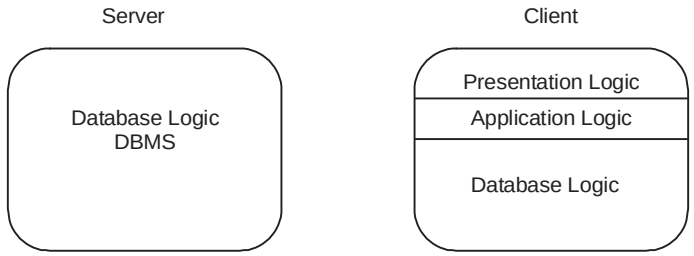


Fig. 5.3: Client-base Processing

(*iv*) Cooperative processing: Taking the advantage of the strengths of both client and server machine and of the distribution of data, the application processing is performed in an optimized fashion. Such a configuration of Client/Server approach is more complex to set up and maintain. But in the long run, this configuration may offer greater user productivity gain and greater network efficiency than others Client/Server approaches. A considerable fraction of the load is on the client, so this is also called *fat client* model. In case of some application development tools this model is very popular shown in Fig. 5.4 given below:

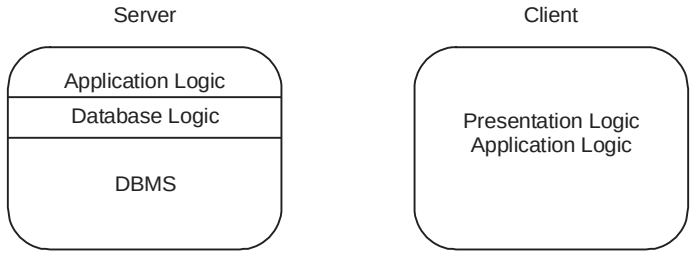


Fig. 5.4: Cooperative Processing

5.5 CLIENT SERVICES

Any workstation that is used by a single user is a client, it has been noticed during last decade the workstations are improving their performance surprisingly. Having same cost you can purchase CPU that can perform more than 50 times, main memory approximately

30 times and Hard Disk up to 40 times. These are considered as power factor of computer, hence, as a result, more sophisticated applications can be run from the workstations. To run various applications workstation uses the available operating systems like DOS, Windows (98, 2000, NT) and UNIX or Linux, Mac, OS/2. In case of, network environment (LAN, WAN) workstations also avails the services provided by the network operating systems. Client workstations request services from the attached server. Whether this server is in fact the same processor or a network processor, the application format of the request is the same. Network operating system translates or adds the specifics required by the targeted requester to the application request. Communication between all these running processes are better described by Inter Process Communication (IPC), these processes might be on the same computer, across the LAN, or WAN.

Some of the main services that client performs (role of client) are listed below:

- Responsible for managing the user interface.
- Provides presentation services.
- Accepts and checks the syntax of user inputs. User input and final output, if any, are presented at the client workstation.
- Acts as a consumer of services provided by one or more server processors.
- Processes application logic.
- The role of the client process can be further extended at the client by adding logic that is not implemented in the host server application. Local editing, automatic data entry, help capabilities, and other logic processes can be added in front of the existing host server application.
- Generates database request and transmits to server.
- Passes response back to server.

But in client server model one thing is very obvious that the services are provided by combination of resources using both the client workstation processor and the server processor. For an example, let us take very common example of client server application, a database server provides data in response to an SQL request issued by the workstation application. Local processing by the workstation might calculate the invoice amount and format the response to the workstation screen. Now, it is important to understand that a workstation can operate as a client in some instances while acting as a server in other instances. For example, in a LAN Manager environment, a workstation might act as a client for one user while simultaneously acting as a print server for many users. In other words we can say "the client workstation can be both client and server when all information and logic pertinent to a request is resident and operates within the client workstation."

Apart from these services discussed above some of the other important services that are directly or indirectly attached with the client services are given below:

(*a*) Inter process communication.

(*b*) Remote services.

(*c*) Window services.

(*d*) Dynamic data exchange.
(*e*) Object linking and embedding.
(*f*) Common object request broker architecture (CORBA).
(*g*) Print/Fax services.
(*h*) Database services.

5.5.1 Inter Process Communication

The communication between two processes take place via buffer. The alternative way of communication is the process of the interprocess communication. The simple mechanism of this is synchronizing their action and without sharing the same address space. This play an important role in the distribute processing environment.

While signals, pipes and names pipes are ways by which processes can communicate. The more redefined method of inter process communication are message queues, semaphores and shared memory. There are four types of mechanisms, involved for such a communications:-

(*i*) Message passing.
(*ii*) Direct communication.
(*iii*) Indirect communication.
(*iv*) Remote procedures call.

(*i*) Message passing: This mechanism allows process to communicate without restoring the shared data, for example in micro kernel, message is passed to communicate in which services acts as an ordinary user where these services act outside the kernel. At least, there are two processes involved in an IPC. See the Fig. 5.5 given below:

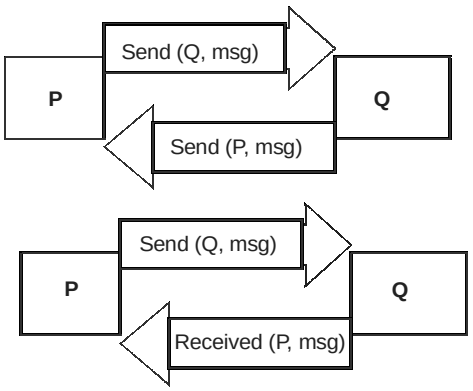


Fig.5.5: Message Passing

- Sending process for sending the message.
- Receiving process for receiving the message.

Messages sent by the processes are of two types, fixed and variable. For the communication to be taking place, a link is to be set in between the two processes.

(*ii*) Direct communication: In this mechanism of communication processes have to specify the name of sender and recipient process name. This type of communication has the following features:

- A link is established in between the sender and receiver along with full known information of their names and addresses.
- One link must be established in between the processes.
- There is symmetry in between the communication of the processes.

(*iii*) Indirect communication: In indirect communication, messages are sending to the mail box and then they are retrieved from mailbox, see the Fig 5.6 given below:

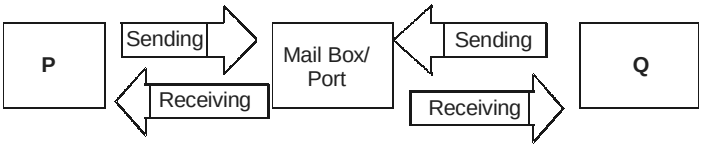


Fig. 5.6: Indirect Communication

The role of the mailbox is quite similar to the role of the postman. The indirect communication can also communicate with other processes via one or more mailbox. The following features are associated with indirect communication:

- A link is established between a pair of process, if they share a mailbox.
- A link is established between more than one processes.
- Different number of links can be established in between the two communicating processes.

Communication between the processes takes place by executing calls to the send and receive primitive. Now there is several different ways to implement these primitives, they can be "blocking" and "non-blocking". The different possible combinations are:

- *Blocking send*: Sending the process is blocked until the message is received.
- *Non-blocking send*: In it process sends the message and then it resumes the operation.
- *Blocking receive:* Receiver is blocked until the message is available.
- *Non-blocking receive:* The receiver receives either a valid message or a null.

(*iv*) Remote procedures call: RPC is a powerful technique for constructing distributed, client-server based applications. The essence of the technology is to allow programs on different machines to interact using simple procedure call or return semantics, just as if the two programs were on the same machine. It is based on extending the notion of conventional or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. That is, the procedure call is used for access to remote services.

In client-server based applications a binding is formed when two applications have made a logical connection and are prepared to exchange commands and data. This client server binding specifies how the relationship between a remote procedure and the calling program will be established. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

How RPC Works: An RPC mechanism is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 5.7 illustrates the general architecture of remote procedure call mechanism that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

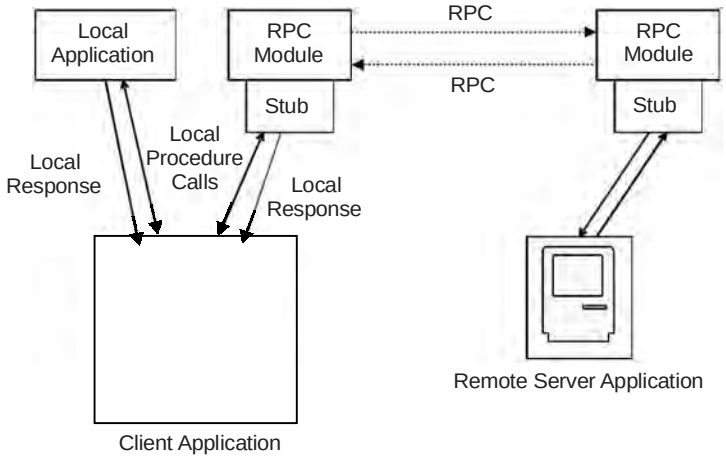


Fig. 5.7: RPC Mechanism

A remote procedure is uniquely identified by the triple: (program number, version number, procedure number), the program number identifies a group of related remote procedures, each of which has a unique procedure number. A program may consist of one or more versions. Each version consists of a collection of procedures which are available to be called remotely. Version numbers enable multiple versions of an RPC protocol to be available simultaneously. Each version contains a number of procedures that can be called remotely. Each procedure has a procedure number. Procedure may or may not be transparent to the user that the intention is to invoke a remote procedure on the same machine.

A stub procedure is added in the callers users address space (or dynamically linked to it at call time). This stub procedure creates a message that identifies the procedure being called and includes the parameters. Stub procedure provides a perfectly local procedure call abstraction by concealing from PROM programs the interface to the underlying RPC system.

RPC provides method for communication between processes residing over a distributed system. Procedure call is used for access to remote services. Basic concepts about this technique is that allowing programs residing on different machines to interact using simple procedures in a similar way like two programs running on the same machine. That is programmers feels an isolation from the network intricacies and got easy access to network functions by using RPC systems services.

RPCs, are APIs, layered on top of a network IPC mechanism, allows users to communicate users directly with each others. They allows individual processing components of an application to run other nodes in the network. Distributed file systems, system management, security and application programming depend on the capabilities of the underlying RPC mechanisms. Server access control and use of a directory service are common needs that can be met with RPCs. RPCs also manage the network interface and handle security and directory services. Tools of RPC comprises of:

- A language and a compiler to produce portable source code.
- Some run time facilities to make the system architecture and network protocols transparent to the application procedure.

The mechanism of RPC can be considered as a refinement of reliable, blocking message passing. Figure 5.8 given below, illustrates the general architecture understanding.

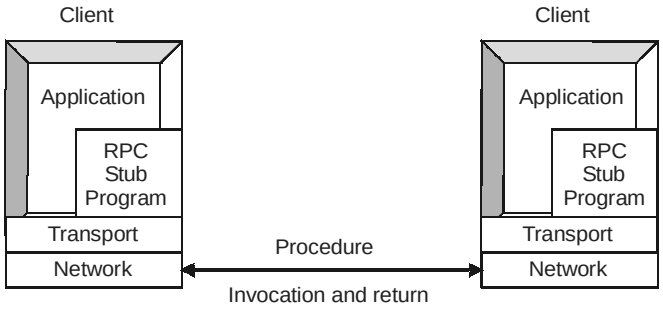


Fig. 5.8: General Architecture

Here, the structure of RPC allows a client to invoke a procedure on the remote host locally, which is done with the help of "stub" which is provided by the client. Thus, when the client invokes the remote procedure RPC calls the appropriate stub, passes the parameters to it, which are then provided, to remote procedure. This stub locates the port on the server and **marshalling** involves packaging the parameter into a form, which may be transmitted over network. The stub then transmits a message to server using message passing. Now the message sent by the host is received at the client side with the help of similar type of stub.

Limitation of RPC: There are number of limitations associated with RPC given below.

1. RPC requires synchronous connections. If an application uses an RPC to link to a server that is busy that time then application will have to wait for the data rather than switching to other task.
2. Local procedure call fails under the circumstances where RPC can be duplicated under and executed more than one, which is due to unreliable communication.
3. The communication in between the client and server is done with help of the standard procedure calls; therefore some binding must take place during the link load and execution, such that the process is replaced by the address. The RPC binds the same thing to the client and server. A general problem that exists is that there is no shared memory in between them so how they can come to know about the address of the other system.
4. The binding information may be predetermined in the form of the port address, at the compile time, a RPC call, that has a fix port number is associated with it. Once a program is compiled, it then cannot change its port number.
5. Binding can be done dynamically by rendezvous mechanism. Typically an operating system provides rendezvous demon requesting the port address of RPC, it needed to execute. The port address is then returned and the RPC call may be sent to the port until the process terminates.

5.5.2 Remote Services

In client server model applications can be invoked directly from the client to execute remotely on a server. The workstation is responsible to provide various remote services. Among them some services like remote login, remote command execution, remote backup services, remote tape drive access and remote boot services, and remote data access are important. Software available with Network Operating System is responsible to run on the client workstation to initiate all these remote services. Client server technology supports full-powered workstations with the capability for GUI applications consistent with the desktop implementation. Remote command execution is when a process on a host cause a program to be executed on another host, usually the invoking process wants to pass data to the remote program, and capture its output also. From a client workstation backup services may be invoked remotely. Some of the business functions such as downloading data from a host or checking a list of stock prices might also be invoked locally to run remotely. To run the application, some of the workstation clients (like X-terminals) do not have the local storage facility. In such scenario, client provides appropriate software's that are burned into E-PROM (Erasable Programmable Read-Only Memory) to start the initial program load (IPL)that is known as Boot Process. If E-PROM is inbuilt with X-terminals to hold the Basic Input/Output System services. Then partial operating system will be able to load the remote software's that provides the remaining services and applications functions to the client workstation. This is known as remote boot service provided by client workstation and X-terminals.

Remote data access is one of the ISO multi-site transaction processing and communication protocol used for heterogeneous data access. Using RDA technology, any client running an application will be able to access more than one database residing at the different servers.

5.5.3 Window Services

In client server application, operating system at the client workstation provides some windows services, these services are capable of to move, view, activate, hide, or size a particular window. This is very helpful in implementation of several applications because a client workstation may have several windows open on-screen at a time. And also, all these applications interact with message services provided to notify the user of events that occur on a server. Application programs running on workstations have been written with no windowing sensitivity. These application programs are written under virtual screen assumptions, that virtual screens are generally dissimilar to the actual available physical screens. Now with the help of interface software client application places data into virtual screen, and then the windows services handles manipulation and placement of application windows. Thus, pursuing that way application development has been enhanced tremendously due to developers less involvement in managing or building the windowing services. The client user is fully in grip of his desktop and can give priority to the most important tasks at hand simply by positioning the window of interest to the workstation.

The NOS provides some software's on the client workstation which is able to manage the creation of pop-up windows that display alerts generated from remote servers. Print complete, E-mail receipt, Fax available, and application termination are examples of alerts that might generate a pop-up window to notify the client user.

5.5.4 Dynamic Data Exchange (DDE)

DDE is usually described as a conversation between two applications, a client application and a server application. As we know that the client program is on that requests (receives) the information, and the server is the one that response (supplies) it. DDE is a feature of some operating systems (like Windows 98, OS/2) presentation manager that enable users to pass data between applications to application. For an example, if an application wants to connect a Microsoft Excel spreadsheet with Microsoft Word for windows report in such a way that changes to the spreadsheet are reflected automatically in the report, in that case Microsoft Word for windows is the client and Microsoft Excel is the server. A DDE conversation always concerns a particular topic and a particular item. The topic and item spell out the nature of the information that the client is requesting from the server. For an example, if the Word for Windows document is to receive data automatically from a range named IBM in a Microsoft Excel worksheet, named STOCKS.XLS then STOCKS.XLS is the topic and IBM is the item. With most of the programs, a simplest way to set up a DDE link is to copy a block of data from the server application to the clipboard, activate the client application, move the insertion point to the location in the receiving document where

you want the information to go, and then use a Paste Link command. With most server programs, some times it requires to save data in a disk file before to paste it into a client programs. Using Paste Link is the easiest way to establish a DDE link, but it's not the only way. Some programs that act as DDE clients have commands that allow you to set up a DDE connection without first putting the source data on the clipboard. Many DDE supporting applications also have macro language that you can use to establish DDE links. This is true with MS Excel, Word, Powerpoint, dynacomm, and many other advanced windows applications. A DDE link may be automatic or manual. An automatic link is refreshed whenever the source data changes, provided both the client and server applications are running. A manual link is refreshed only when you issue a command in the client application.

5.5.5 Object Linking and Embedding (OLE)

Object Linking and Embedding two services collectively called as a single one, carried out with simple edit menu procedures. OLE is a software package accesses data created from another through the use of a *viewer* or *launcher*. These viewers and launchers must be custom built for every application. With the viewer, users can see data from one software package while they are running another package. Launchers invoke the software package that created the data and thus provide the full functionality of the launched software. To link with OLE copy data from OLE supporting program to the Clipboard. Then use the paste link command in another OLE supporting program. To embed, follow the same procedure but use Paste instead of Paste Link. Both programs must support OLE, the program that supplies the data must support OLE as a server application, and the one that receives the data must support as a client application. Some program may support OLE in one mode or the other, (it means either as a server or as client only). For an example, Paintbrush can act only as a server. Write and Card file can act only as OLE clients. Some other programs are also available which support OLE in both modes. Most of the Windows applications support OLE, and also the Microsoft has released its OLE 2.0 Software Development Kit (SDK). The toolkit greatly simplifies OLE integration into third-party, developed applications. Organizations wanting to create a consistent desktop are beginning to use the OLE SDK as part of custom applications. OLE 2.0 extends OLE capabilities to enable a group of data to be defined as an object and saved into a database. This object can then be dragged and dropped into other applications and edited without the need to switch back to the application which created it. This provides a more seamless interface for the user. In OLE 2.0, the active window menu and toolbar change to that of 1-2-3. The user deals only with the object, with no need to be aware of the multiple software being loaded. Generally, the OLE is known as an extension to DDE that enables objects to be created with the object components software aware *(*a reference to the object or one of its components automatically launches the appropriate software to manipulate the data). Both the techniques (OLE and DDE) require the user to be aware of the difference between

data sources. DDE and OLE provide a substantial advantage; any DDE-or OLE-enabled application can use any software that supports these data interchange APIs. An e-mail application will be able to attach any number of components into the mail object without the need to provide custom viewers or launchers. But here, linking with OLE offers one big advantage over linking with DDE; that is the ability to launch the server program directly forms the client document. Client need not to remember from where the data came from. And if the server renames or relocates a document it requires repairing or abandoning any links in client documents. Most OLE client's applications include commands to assist such an application with this.

5.5.6 Common Object Request Broker Architecture (CORBA)

CORBA an object oriented architecture that provides an mechanism that allows various clients to share/call the object (applications) over a mixed networks. More specifically CORBA is a process of moving objects over network proving cross platform for data transfer. A client that needs a service sends a request to an object request broker (which acts as a directory) of all the remote services available on the network, illustrated in Fig. 5.9 given below:

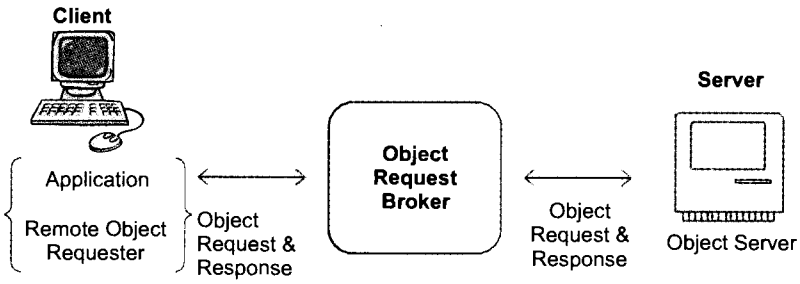


Fig. 5.9: Object Request Broker

The broker calls the appropriate object and passes along any relevant data. Then the remote object services the request and replies to the broker, which returns the response to the client. The object communication may rely on an underlying message or RPC structure or be developed directly on top of object-oriented capability in the operating system. The architecture of CORBA application is similar to the client server architecture by maintaining the notion of client and servers. In CORBA, a component can act as both a client and server. A component is considered as a server if it contains CORBA objects whose services are accessible form some other CORBA object. Likewise, a component is considered as a client if it access services from some other CORBA objects. Here a component can simultaneously provides and use various services and so any component can be considered as a client or as a server depending on the nature of the application running currently. When considering a single remote method invocation, however the role of client and server can be temporarily reversed because a CORBA object can participate in multiple interactions simultaneously.

Furthermore, any component that provides an implementation for an object is considered as a server, at least where that object is concerned. If a component creates an object and provides other components with visibility to that object (i.e., allows other components to obtain references to that object), that components acts as server for that object; any requests made on that object by other components will be processed by the components that creates the object. Thus, a CORBA server means the components execute methods for a particular object on behalf of other components (Clients). An application component can provide services to other application components while accessing services from other components. In that scenario, the components is acting as a client of one component and as a server to the other components i.e., two components can simultaneously act as client and server to each other, illustrated in Fig. 5.10, shown below. CORBA concepts and its role in Client/Server architecture is discussed in more detail in Chapter 9.

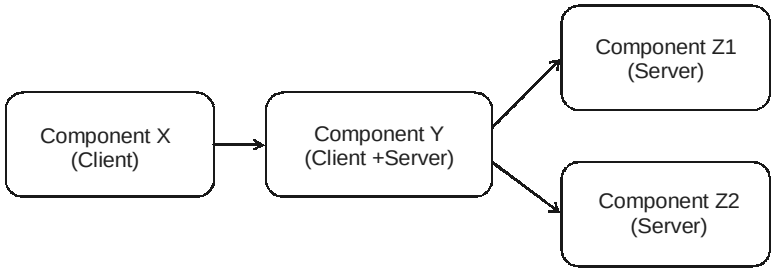


Fig. 5.10: *Acting as a Client and a Server*

5.5.7 Print/Fax Services

Client generates print/fax requests to the printer/fax machine without knowing whether they are free or busy. In that task network operating system helps the client to generate the requests. These requests are redirected by the NOS redirector software and managed by the print/fax server queue manager. The users at the client workstation can view the status of the print/fax queues at any time. And also some of the print/fax servers acknowledge the client workstation when the print/fax request is completed.

5.5.8 Database Services

Client/Server model provides integration of data and services allow clients to be isolated from inherent complexities such as communication protocols. The simplicity of client server architecture allows clients to make request that are routed to the database server (These requests are made in the form of transactions). In other words, client application submit database request to the server using SQL statements. Once received the server processes the SQL statement and the request are returned to the client application. That is illustrated in Fig. 5.11.

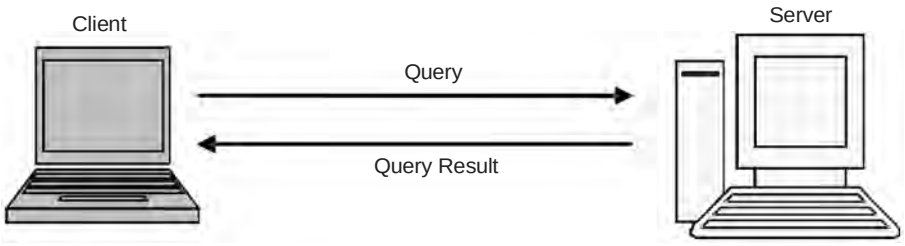


Fig. 5.11: Execution of SQL

Hence, most of the database requests are made using the SQL syntax. Now, the SQL's have became the industry standard language supported by many vendors. Because the language uses a standard form, the same application may be run on multiple platforms. Client application can concentrate on requesting input from users, requesting desired data from server, and then analyzing and presenting this data using the display capabilities of the client workstation. Furthermore, client applications can be designed with no dependence on the physical location of the data. If the data is moved or distributed to the other database servers, the application continues to function with little or no modification. Client applications can be optimized for the presentation of data and server can be optimized for the processing and storage of data. Application development tools are used to construct the user interfaces (interface of clients with server and also interface of front-end user to the back-end server); they provide graphical tools that can be used to construct interfaces without any programming. Examples of such tools are Visual Basic, Borland Delphi, Magic, and Power Builder. Some application programs (spreadsheet and statistical–analysis packages) uses the client server interface directly to access data from back-end server.

5.6 SERVER SERVICES

The server is responsible for controlling and providing shared access to available server resources. Remote workgroups have needed to share these resources when they are connected with server station through a well-managed network. The applications on a server must be isolated from each other so that an error in one application cannot damage another application. Furthermore, the server is responsible for managing the server-requester interface so that an individual client request response is synchronized and directed back only to the client requester. This implies both security when authorizing access to a service and integrity of the response to the request. For a Client/Server applications servers performs well when they are configured with an operating system that supports shared memory, application isolation, and preemptive multitasking (an operating system with preemptive multitasking enables a higher priority task to preempt or take control of the processor from a currently executing, lower priority task). These preemptive multitasking ensures that no single task can take overall the resources of the server and prevent other

tasks from providing service. There must be a means of defining the relative priority of the tasks on the server. These are specific requirements to the Client/Server implementation.

One of the prime server characteristic is that it must support for multiple simultaneous client request for service. So that, the server must provide shared memory services and multitasking support. In that respect, the following server platform provides best processors for client server implementation are IBM System/370, DEC VAX , Intel and RISC (Reduced Instruction Set Computers like Sun SPARC, IBM/Motorola PowerPC, HP PA RISC, SGI MIPS, and DEC Alpha). Some of the main operations that server perform are listed below:

- Accepts and processes database requests from client.
- Checks authorization.
- Ensure that integrity constraints are not violated.
- Performs query/update processing and transmits response to client.
- Maintains system catalog.
- Provide concurrent database access.
- Provides recovery control.

Apart from these services discussed above some of the other important services that are directly or indirectly attached with the server services in a network operating system environment are given below:

(*i*) Application services.
(*ii*) File services.
(*iii*) Database services.
(*iv*) Print/fax/image services.
(*v*) Communications services.
(*vi*) Security systems services.
(*vii*) Network management services.
(*viii*) Server operating system services.

(*i*) Application services: Application servers provide business services to support the operation of the client workstation. In the Client/Server model these services can be provided for entire partial business functions that are invoked by IPC (Inter Process Communication) or RPCs request for service. A collection of application servers may work in concert to provide an entire business function. For an example, in a inventory control system the stock information may be managed by one application server, sales information are maintained by another application server, and purchase information are maintained by a third application server. On larger and more complicated systems, server responsibility may be distributed among several different types of servers. All these servers are running at different operating systems on various hardware platforms and may use different database servers. The

client application invokes these services without consideration of the technology or geographic location of the various servers.

(*ii*) **File services:** A file server can store any type of data, and so on simpler systems, may be the only server necessary. Space for storage is allocated, and free space is managed by the file server. Catalog functions are also provided by the file server to support file naming and directory structure.

File services are responsible to handle access to the virtual directories and files located on the client workstation and to the server's permanent storage. Redirection software provides these services which are installed as part of client workstation operating system. Finally, all clients' workstation requests are mapped into the virtual pool of resources and redirected as necessary to the appropriate local or remote server. The file services provide this support at the remote server processor. File server manages databases, software's, shared data, and backups that are stored on tape, disk, and optical storage devices. In order to minimize the installation and maintenance effort of software, software should be loaded directly from the server for execution on the client workstations. New versions of any application software can be updated on the server and made immediately available to all users.

(*iii*) **Database services:** Early database servers were actually file servers with a different interface. Products such as dBASE, Clipper, FoxPro, and Paradox execute the database engine primarily on the client machine and use the file services provided by the file server for record access and free space management. These are new and more powerful implementations of the original flat-file models with extracted indexes for direct record access. Currency control is managed by the application program, which issues lock requests and lock checks, and by the database server, which creates a lock table that is interrogated whenever a record access lock check is generated. Because access is at the record level, all records satisfying the primary key must be returned to the client workstation for filtering. There are no facilities to execute procedural code at the server, to execute joins, or to filter rows prior to returning them to the workstation. This lack of capability dramatically increases the likelihood of records being locked when several clients are accessing the same database and increases network traffic when many unnecessary rows are returned to the workstation only to be rejected.

The lack of server execution logic prevents these products from providing automatic partial update backout and recovery after an application, system, or hardware failure. For this reason, systems that operate in this environment require an experienced system support programmer to assist in the recovery after a failure. When the applications are very straight forward and require only a single row to be updated in each interaction, this recovery issue does not arise. However, many Client/Server applications are required to update more than a single row as part of one logical unit of work.

Client/Server database engines such as Sybase, IBM's Database Manager, Ingres, Oracle, and Informix provide support at the server to execute SQL requests issued from the client workstation. The file services are still used for space allocation and basic directory services, but all other services are provided directly by the database server. Relational database management systems are the current technology for data management. Figure 4.1 charts the evolution of database technology from the first computers in the late 1950s to the object-oriented database technologies that are becoming prevalent in the mid-1990s.The following DBMS features must be included in the database engine:

- Performance optimization tools.
- Dynamic transaction backout.
- Roll back from, roll forward to last backup.
- Audit file recovery.
- Automatic error detection and recovery.
- File reclamation and repair tools.
- Support for mirrored databases.
- Capability to split database between physical disk drives.
- Remote distributed database management features.
- Maintenance of accurate and duplicate audit files on any LAN node.

In the Client/Server implementation, database processing should offload to the server. Therefore, the database engine should accept SQL requests from the client and execute them totally on the server, returning only the answer set to the client requestor. The database engine should provide support for stored procedures or triggers that run on the server.

The Client/Server model implies that there will be multiple concurrent user access. The database engine must be able to manage this access without requiring every developer to write well-behaved applications. The following features must be part of the database engine:

- Locking mechanisms to guarantee data integrity.
- Deadlock detection and prevention.
- Multithreaded application processing
- User access to multiple databases on multiple servers.

(*iv*) Print/fax/image services: High-quality printers, workstation-generated faxes, and plotters are natural candidates for support from a shared server. The server can accept input from many clients, queue it according to the priority of the request and handle it when the device is available. Many organizations realize substantial savings by enabling users to generate fax output from their workstations and queue it at a fax server for transmission when the communication costs are lower. Incoming faxes can be queued at the server and transmitted to the appropriate client either on receipt or on request. In concert with workflow management techniques, images can be captured and distributed to the appropriate client

workstation from the image server. In the Client/Server model, work queues are maintained at the server by a supervisor in concert with default algorithms that determine how to distribute the queued work.

Incoming paper mail can be converted to image form in the mail room and sent to the appropriate client through the LAN rather than through interoffice mail. Centralized capture and distribution enable images to be centrally indexed. This index can be maintained by the database services for all authorized users to query. In this way, images are captured once and are available for distribution immediately to all authorized users. Well-defined standards for electronic document management will allow this technology to become fully integrated into the desktop work environment. There are dramatic opportunities for cost savings and improvements in efficiency, if this technology is properly implemented and used. Chapter 9 discusses in more detail the issues of electronic document management.

(*v*) **Communications services:** Client/server applications require LAN and WAN communication services. Basic LAN services are integral to the NOS. WAN services are provided by various communications server products. Chapter 5 provides a complete discussion of connectivity issues in the Client/Server model.

(*vi*) **Security systems services:** Client/server applications require similar security services to those provided by host environments. Every user should be required to log in with a user ID and password. If passwords might become visible to unauthorized users, the security server should insist that passwords be changed regularly. The enterprise on the desk implies that a single logon ID and logon sequence is used to gain the authority once to access all information and process for the user has a need and right of access. Because data may be stored in a less physically secure area, the option should exist to store data in an encrypted form. A combination of the user ID and password should be required to decrypt the data.

New options, such as floppy less workstation with integrated Data Encryption Standard (DES) coprocessors, are available from vendors such as Beaver Computer Company. These products automatically encrypt or decrypt data written or read to disk or a communication line. The encryption and decryption are done using the DES algorithm and the user password. This ensures that no unauthorized user can access stored data or communications data. This type of security is particularly useful for laptop computers participating in Client/Server applications, because laptops do not operate in surroundings with the same physical security of an office. To be able to access the system from a laptop without properly utilizing an ID number and password would be courting disaster.

5.7 CLIENT/SERVER APPLICATION: CONNECTIVITY

The communication middleware software provides the means through which clients and servers communicate to perform specific actions. It also provides specialized services to

the client process that insulate the front-end applications programmer from the internal working of the database server and network protocols. In the past, applications programmers had to write code that would directly interface with specific database language (generally a version of SQL) and the specific network protocol used by the database server. For example, when writing a front-end application to access an IBM OS/2 database manager database, the programmer had to write SQL and Net BIOS (Network Protocol) command in the application. The Net BIOS command would allow the client process to establish a session with the database server, send specific control information, send the request, and so on. If the same application is to be use with a different database and network, the application's routines must be rewritten for the new database and network protocols. Clearly such a condition is undesirable, and this is where middleware comes in handy. Here definition of middleware is based on the intended goals and main functions of this new software category. In chapter three communication middleware is also discussed, further role and mechanism of middleware is discussed in this section.

5.7.1 Role and Mechanism of Middleware

Role of middleware component can be exactly understand by the way in which Client/Server computing being used , we know that there are number of approaches are there like host-based processing, server based processing, cooperative processing and client based processing. And all these depend on application functionality being used in Client/Server architecture. A Middleware component resides on both Client/Server machine enabling an application or user at a client to access a variety of services provided by server.

In other words, we can say middleware provides basis for logical view of Client/Server architecture. Morever, middleware enables the realization of the promises of distributed Client/Server computing concepts. See the Fig. 5.12 (*a*) and (*b*) that depicts the role and logical view of middleware in Client/Server architecture. The entire system of the architecture represents a view of a set of applications and resources available to clients. Any client need not be concerned with the location of data or indeed the location of the application.

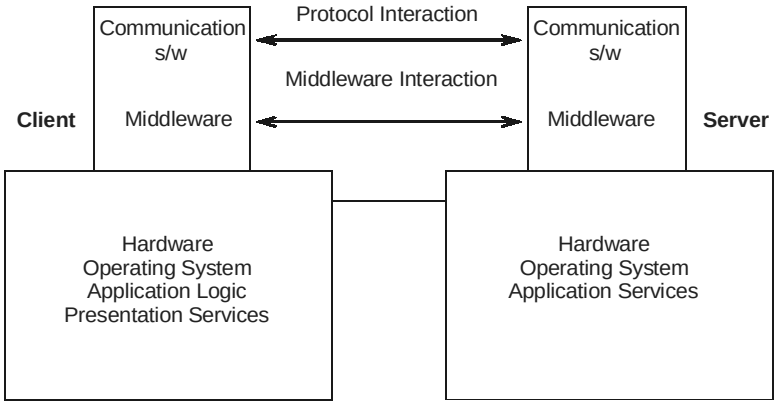


Fig. 5.12(a): Middleware Role in Client/Server Architecture

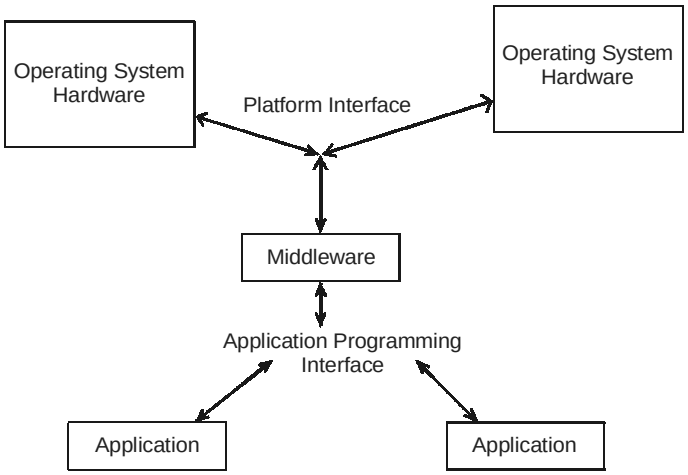


Fig. 5.12(*b*): Middleware Role in Client/Server Architecture

All applications operate over a uniform application programming interface. The middleware is responsible for routing client requests to the appropriate server. Also middleware used to overcome operating system as well as network incompatibility. This middleware running on each network component ensures that all network users have transparent access to applications and resources of any networks.

5.8 CLIENT/SERVER APPLICATION: LAYERED ARCHITECTURE

An essential factor in the success of a client/server environment is the way in which the user interacts with the system as a whole. Thus the design of the user interface to the client machine is critical. In most client/server systems, there is heavy emphasis on providing a graphical user interface that is easy to use, easy to learn, yet powerful and flexible. Section follow covers the design issues of client/server architecture. And also designing issues associated with layered application architecture with their interfaces in the three-layered application architecture.

5.8.1 Design Approach

In client/server architecture as a design approach, the functional components of an application are partitioned in a manner that allows them to be spread and executed across different computing platforms, and share access to one or more common repositories. Client/server architecture is therefore a design approach that distributes the functional processing of an application across two or more different processing platforms. The phrase 'client/server' reflects the role played by an application's functions as they interact with one another. One or more of these functions is to provide a service, typically in the form of a database server that is commonly used by other functions across the application(s). In this regard, it is important to discuss the concept of 'layered application architecture' .

Application design architecture plays a crucial role in aiding development of robust applications. Figure. 5.13 given below shows the three layers of any fairly large business

application. The most detailed layer is the database layer, the centre of an application. The next higher layer is the business rule layer. Finally, the highest level of abstraction is the document layer. This is the layer visible to users of the application.

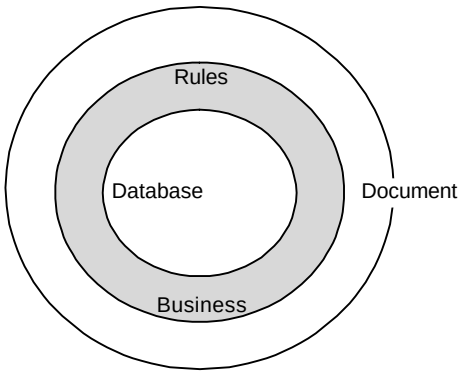


Fig. 5.13: Three-layered Application Architecture

Designing a client/server application offers challenges not normally faced by developers of mainframe-oriented multiuser applications. To realize full benefits of client/server architecture, developers need to incorporate a greater potential for functionality and data distribution in the fundamental design of their applications.

A client/server application operates across multiple platforms, i.e. a server platform for the database, and a client platform for the application. At this minimum level of utilization, the design of client/server does not differ much from its mainframe counterpart, and the physical issues faced by developers on both the platforms are primarily those of packaging. But to take full advantage of client/server paradigm, developers need to address issues of functional distribution for their applications, not just across client and server platform but also across all nodes of the network. Issues surrounding functional distribution constitute the single biggest difference between physical designs of multiuser and client/server application.

5.8.2 Interface in Three Layers

The key to use a three-layered application architecture is to understand the interfaces used in all its three layers. Figure 5.14 illustrates these interfaces. They are:

- Graphical user interface.
- Process request interface.
- Transaction and query manager interface.

An interface enables a component in one layer to communicate with a component in another layer; it also enables a component to interact with another component in the same layer. In Fig. 5.14 communication between components in the same layer is indicated by a semicircular arrow. Moreover, we can say that this describes a collection of mutually cooperating components that make request to each other, both within and across layers, with its components working together thus, and processing various requests—wherever they comes from; a business application comes to life.

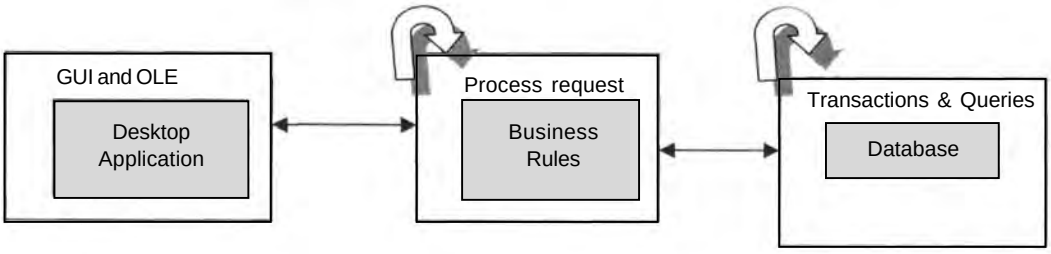


Fig. 5.14: Interface in a Three-layered Application Architecture

Cooperating components in a layered application design provide the following:

- A framework for building highly flexible applications that can be changed easily to meet the changing needs of business.
- A high level of software reuse.
- Easier development of large, complex applications that can sustain high throughput levels in both decision support and transaction environments.
- Easier development of distributed applications that support centrally and self-managed teams.

EXERCISE 5

1. In Client/Server computing, explain the following with example in detail
   (*a*) Dynamic Data Exchange
   (*b*) RPC, Remote Procedure Call
   (*c*) Remote Boot Service
   (*d*) Diskless Computer
   (*e*) Object-linking and embedding
2. Explain the role of client in Client/Server computing and also explain the various services provide by client.
3. Explain the server functionality, in detail, for Client/Server computing.
4. What is Interring Process Communication (IPC) and what are services provided by IPC? Also explain various protocol used for IPC.
5. What was the primary motivation behind the development of the RPC facility? How does a RPC facility make the job of distributed applications programmers simpler?
6. Explain basic Interprocess Communication Mechanism. Explain Port Management and Message Passing in IPC.
7. What are the main similarities and differences between the RPC model and the ordinary procedure call model?
8. Why do most RPC system support call by value semantics for parameter passing?
9. Explain the difference between the terms service and server.
10. Explain the role of server in Client/Server computing and also explain the various services provided by server.

6

System Development

6.1 HARDWARE REQUIREMENTS

6.1.1 PC Level Processing Units

UNIX Workstations

The user running Client/Server applications form DOS or Windows typically run only a single business process at a time. And also UNIX has locked the familiar personal productivity tools such as word processors, e-mail, spreadsheet, presentation graphics and database management system, but recently few personal productivity applications were in place, user needs have increased with providing reliability with multitasking. Many Unix implementation with application execution offers the best of all words for the desktop user reliability and functionality. Nowadays Unix supports many of the most familiar personal computer applications like WordPerfect, DBASE IV, Lotus 1-2-3. Unix has become the workstation of choice for Client/Server environment on the basis of cost performance rather than functionality.

X-Window System

The X-Window System is an open, cross-platform, Client/Server system for managing a windowed graphical user interface in a distributed network. In X-Window, the Client/Server relationship is reversed from the usual. Remote computers contain applications that make client requests for display management services in each PC or workstation. X-Window is primarily used in networks of interconnected mainframes, minicomputers, and workstations. It is also used on the X-terminal, which is essentially a workstation with display management capabilities but without its own applications. (The X-terminal can be seen as a predecessor of the network PC or thin client computer).

X-Window System (commonly X11 or X) is a windowing system for bitmap displays. It provides the standard toolkit and protocol to build graphical user interfaces on Unix, Unix-like operating systems, and OpenVMS; and almost all modern operating systems support it. X provides the basic framework for a GUI environment to do drawing and moving windows on the screen and interacting with a mouse and keyboard. X does not mandate the user interface, individual client programs handle this. As such, the visual styling of X-based environments varies greatly; different programs may present radically different interfaces. X provides network transparency in which the machine where application programs (the *client* applications) run can differ from the user's local machine (the display *server*).

X-Terminal

An X-terminal is typically a diskless terminal especially designed to provide a low-cost user interface for applications that run in a network X-server as part of a distributed X-Window System. Typically, X-terminals are connected to a server running a UNIX-based operating system in a mainframe, minicomputer, or workstation. A terminal specially designed to run an X-server which allows users to display the output of programs running on another computer using the X-protocol over a network.

The X-terminal concept is essentially like tel-neting into a machine and then running some application there. All the working is done on the machine that you are connecting to but the display is shown on your machine. That just gives you access to console mode text applications, whereas an X-terminal setup will give you access to the entire range of GUI applications. All applications will be run on the server but the display will be exported to your computer. The machine that you setup as the X-terminal just serves as a display. This setup works very well with diskless workstations and older computers. An X-terminal is a great way to expand the computing presence in a home or office.

An X-terminal consists of a piece of dedicated hardware running an X-server as a thin client. This architecture became popular for building inexpensive terminal parks for many users to simultaneously use the same large server. X-terminals can explore the network (the local broadcast domain) using the X-Display Manager Control Protocol to generate a list of available hosts that they can run clients from. The initial host needs to run an X-display manager. Dedicated (hardware) X-terminals have become less common; a PC with an X-server typically provides the same functionality at a lower cost.

X-Server

An X-server is a server of connections to X-terminal in a distributed network that uses the X-Window System. From the terminal user's point-of-view, the X-server may seem like a server of applications in multiple windows. Actually, the applications in the remote computer with the X-server are making client request for the services of a windows manager that runs in each terminal. X-servers (as part of the X-Window System) typically are installed in a UNIX-based operating system in a mainframe, minicomputer, or workstation.

The X-server is the software that handles all the interactions between the GUI and hardware used. Windows equivalent would be the graphics card driver. But X is a lot more than that. Here it becomes a server with whom clients get connected. Clients would be the various GUI applications like GNOME, KDE etc. communicating through network protocols. This architecture allows a lot of flexibility. The clients can be run on any machine but the display can be routed to another machine. The X-server provides the following services.

- *Window services:* Clients ask the server to create or destroy windows, to change their attributes, to request information about them, etc.
- *Input handling:* Keyboard and mouse input are detected by the server and sent to clients.
- *Graphic operations:* Clients ask the server to draw pixels, lines, strings, etc. The client can ask information about fonts (size, etc.) and can ask transfer of graphic content.
- *Resource management:* The X-resource manager provides a content addressable database for clients. Clients can be implemented so they are customizable on a system and user basis.

The X-Client/Server model and network transparency

In X-Client/Server model, an *X-server* communicates with various *client* programs. The server accepts requests for graphical output (windows) and sends back user input (from keyboard, mouse, or touchscreen). The server may function as any one of the following:

- an application displaying to a window of another display system.
- a system program controlling the video output of a PC.
- a dedicated piece of hardware.

This Client/Server terminology the user's terminal as the "server", the remote applications as the "clients" often confuses new X users, because the terms appear reversed. But X takes the perspective of the program, rather than the end-user or the hardware. The local X display provides display services to programs, so it is acting as a server; the remote program uses these services, thus it acts as a client.

In above example, the X-server takes input from a keyboard and mouse and displays to a screen. A web browser and a terminal emulator run on the user's workstation, and a system updater runs on a remote server but is controlled from the user's machine. Note that the remote application runs just as it would locally.

The communication protocol between server and client operates network-transparently. The client and server may run on the same machine or on different ones, possibly with different architectures and operating systems, but they run the same in either case. A client and server can even communicate securely over the Internet by tunneling the connection over an encrypted connection. To start a remote client program displaying to a local server, the user will typically open a terminal window and telnet or ssh to the remote

machine, tell it to display to the user's machine (*e.g.,* export DISPLAY = *[user's machine]*:0 on a remote machine running bash), then start the client. The client will then connect to the local server and the remote application will display to the local screen and accept input from the local input devices. Alternately, the local machine may run a small helper program to connect to a remote machine and start the desired client application there. Practical examples of remote clients include:

- administering a remote machine graphically.
- running a computationally-intensive simulation on a remote Unix machine and displaying the results on a local Windows desktop machine.
- running graphical software on several machines at once, controlled by a single display, keyboard and mouse.

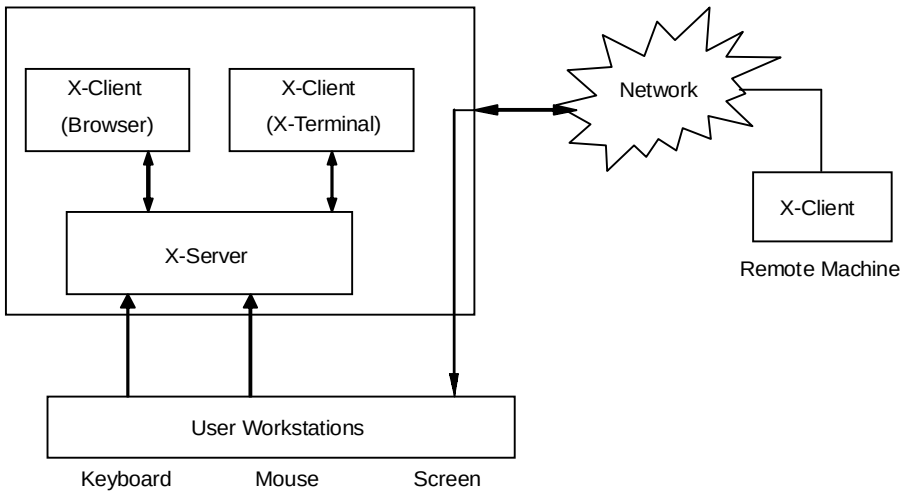


Fig. 6.1: X-Client-Server Model

Light Pen

Light Pen is an input device that utilizes a light-sensitive detector to select objects on a display screen. It is similar to a mouse, except that with a light pen you can move the pointer and select objects on the display screen by directly pointing to the objects with the pen. A light pen is pointing device that can be used to select an option by simply pointing at it, drawing figures directly on the screen. It has a photo-detector at its tip. This detector can detect changes in brightness of the screen. When the pen is pointed at a particular spot on the screen, it records change in brightness instantly and inform the computer about this. The computer can find out the exact spot with this information. Thus, the computer can identify where you are pointing on the screen.

Light pen is useful for menu-based applications. Instead of moving the mouse around or using a keyboard, the user can select an option by pointing at it. A light pen is also useful for drawing graphics in CAD.

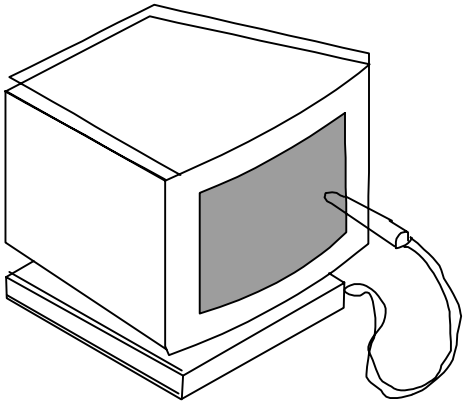


Fig. 6.2: Light Pen

Digital Pen

A digital pen writes on paper like any normal pen. The difference is that it captures everything you write. The digital pens include a tiny camera, some memory, a CPU and a communications unit. The paper is also special in that it needs to have an almost invisible dot pattern printed on it. You could use your laser to print this or get a specialist stationery printer to do it. Many paper products from 3M yellow sticky notes to black n' red notebooks are already available with the pattern pre-printed on them. The pen senses the pattern and this is how it knows where on the page you are writing. Most importantly using the digital pen is as easy as a normal pen with the quite significant benefit that a digital record is simultaneously created as you write.

They are available with desktop software applications integrating the pen with Microsoft Word and Outlook as well as a searchable notebook application. The pen is able to sent what you have written to a computer for storage and processing, or as an e-mail or fax. Applications range from: removing the need to re-key forms, to automatically storing and indexing pages written in a notebook. You can even send faxes and emails by simply writing them with a pen. Example of digital pens is Logitech io2 or a Nokia SU-1B pen.

Notebook Computers

If the portable computers are classified, they are of three types: laptops, notebooks and palmtops. Notebook computers are about the size of a notebook (approx. 21* 29.7 cm) and weight about 3 to 4 kg. Notebooks also offer the same power as a desktop PC. Notebooks have been designed to overcome the disadvantage of laptops that is they are bulky. Notebook/Portable computers are productivity-enhancement tools that allow busy execution to carry their office work with them. They are smaller in size. Several innovative techniques are being used to reduce size. Like VDU is compact, light, and usesless power, LCD (liquid crystal display that are light and consume very little power are used. Further

numbers of keys on keyboard are reduced and also they are made to perform multiple functions. The size of hard disk is reduced is of 2.5" in diameter but capable of storing large quantities of data with weight only 300 gms. Examples of notebooks are Conture 3/20 from Compaq, and AcerAnyWhere from Zenith Computers.

6.1.2 Storage Devices

Storage refers to the media and methods used to keep information available for later use. Some things will be needed right away while other won't be needed for extended periods of time. So different methods are appropriate for different uses. Auxiliary Storage that is Secondary Storage holds what is not currently being processed. This is the stuff that is "filed away", but is ready to be pulled out when needed. It is non-volatile, meaning that turning the power off does not erase it. Auxiliary Storage is used for:

- Input—data and programs.
- Output—saving the results of processing.

So, Auxiliary Storage is where you put last year's tax info, addresses for old customers, programs you may or may not ever use, data you entered yesterday - everything that is not being used right now.

- Magnetic tape.
- Magnetic disks.
- Optical disks.
- Other storage devices—flash drives.

Magnetic Tape

Magnetic tape is a secondary storage device, generally used for backup purposes. They are permanent and not volatile by nature. The speed of access can be quite slow, however, when the tape is long and what you want is not near the start. So this method is used primarily for major backups of large amounts of data. Method used to store data on magnetic tape is similar to that of VCR tape. The magnetic tape is made up of mylar (plastic material) coated only on one side of the tape with magnetic material (Iron oxide). There are various types of magnetic tapes are available. But each different tape storage system has its own requirements as to the size, the container type, and the magnetic characteristics of the tape. Older systems designed for networks use reel-to-reel tapes. Newer systems use cassettes. Some of these are even smaller than an audio cassette but hold more data that the huge reels. Even if they look alike, the magnetic characteristics of tapes can vary. It is important to use the tape that is right for the system. Just as floppy disks and hard disks have several different formats, so do magnetic tapes. The format method will determine the some important characteristics like

Density: Higher density means more data on shorter tape that is measured as bpi (bits per inch that ranges from 800 bpi up to 6250 bpi.

Block: The tape is divided into logical blocks, as a floppy is divided into tracks and sectors. One file could take up many logical blocks, but must take up one whole block at least. So smaller blocks would result in more room for data.

Gap: Two kinds of blank spots, called gaps, are set on the tape. Interblock gap, which separates logical blocks. Interrecord gap, which is wider and separates records. Notice the two size lines cutting across the tape in the Fig. 6.3 below. Smaller gaps would allow more data to be stored on the same size tape.

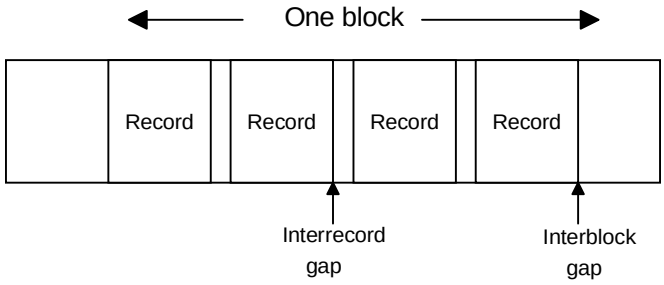


Fig. 6.3: Magnetic Tape

Magnetic Disks

There are various types of auxiliary storage; all of them involve some type of magnetic disk. These come in various sizes and materials, as we shall see. This method uses magnetism to store the data on a magnetic surface. The advantages associated with such type of storage media is they are of high storage capacity, reliable and provides direct access to the data. A drive spins the disk very quickly underneath a *read/write head*, which does what its name says. It reads data from a disk and writes data to a disk.

There are various types of auxiliary storage; all of them involve some type of magnetic disk. These come in various sizes and materials. This method uses magnetism to store the data on a magnetic surface. The advantages associated with such type of storage media is they are of high storage capacity, reliable and provides direct access to the data. A drive spins the disk very quickly underneath a *read/write head*, which does what its name says. It reads data from a disk and writes data to a disk. The available magnetic disks are Diskette/ Floppy disk and Hard disk.

All the magnetic disks are similarly formatted, or divided into areas that are tracks sectors and cylinders. The formatting process sets up a method of assigning addresses to the different areas. It also sets up an area for keeping the list of addresses. Without formatting there would be no way to know what data went with what. It would be like a library where the pages were not in books, but were scattered around on the shelves and tables and floors.

All the magnetic disks contain a track that is a circular ring on one side of the disk. Each track has a number. A disk sector is a wedge-shape piece of the disk. Each sector is numbered. Generally on a 5¼″ disk there are 40 tracks with 9 sectors each and on a 3½″ disk there are 80 tracks with 9 sectors each. Further a track sector is the area of intersection of a track and a sector. A cluster is a set of track sectors, ranging from 2 to 32 or more, depending on the formatting scheme in use.

The most common formatting scheme for PCs sets the number of track sectors in a cluster based on the capacity of the disk. A 1.2 giga hard drive will have clusters twice as large as a 500 MB hard drive. One cluster is the minimum space used by any read or write. So there is often a lot of slack space, unused space, in the cluster beyond the data stored there. The only way to reduce the amount of slack space is to reduce the size of a cluster by changing the method of formatting. You could have more tracks on the disk, or else more sectors on a track, or you could reduce the number of track sectors in a cluster.

A cylinder is a set of matched tracks on a double-sided floppy, a track from the top surface and the same number of track from the bottom surface of the disk make up a cylinder. The concept is not particularly useful for floppies. On a hard disk, a cylinder is made of all the tracks of the same number from all the metal disks that make up the "hard disk." If all these are putted together on the top of each others. It will looks like a tin can with no top or bottom forming a cylinder.

What happens when a disk is formatted?

Whether all data is erased? Surfaces are checked for physical and magnetic defects. A root directory is created to list where things are on the disk.

The capacity of a magnetic disk depends on several factors.

Optical Disk

The disk is made up of a resin (such as polycarbonate) coated with a highly reflective material (Aluminium and also silicon, silver, or gold in double-layered DVDs). The data is stored on a layer inside the polycarbonate. A metal layer reflects the laser light back to a sensor. Information is written to read from an optical disk using laser beam. Only one surface of an optical disk is used to store data. The coating will change when a high intensity laser beam is focused on it. The high intensity laser beam forms a tiny pit along a trace to represent 1 for reading the data laser beam of less intensity is employed (normally it is 25mW for writing and 5mW for reading). Optical disks are inexpensive and have long life up to 100 years. The data layer is physically molded into the polycarbonate. Pits (depressions) and lands (surfaces) form the digital data. A metal coating (usually aluminium) reflects the laser light back to the sensor. Oxygen can seep into the disk, especially in high temperatures and high humidity. This corrodes the aluminium, making it too dull to reflect the laser correctly. There are three types of optical disk are available:

- Compact Disk Read Only Memory (CD-ROM)
- Write Once Read Many (WORM)
- Erasable Optical Disk
- Digital Video Device (DVD)

All these optical disk are of similar characteristics like formed layers, organization of data in a spiral groove on starting form the center of the disk and finally nature of stored data is digital. 1's and 0's are formed by how the disk absorbs or reflects light from a tiny laser. An option for backup storage of changing data is **rewritable disks,** CD-RW, DVD-

RW, DVD + RW, and DVD + RAM. The data layer for these disks uses a phase-changing metal alloy film. This film can be melted by the laser's heat to level out the marks made by the laser and then lasered again to record new data. In theory you can erase and write on these disks as many as 1000 times, for CD-RW, and even 100,000 times for the DVD-RW types.

In case of WORM, the user can write data on WORM and read the written data as many times desired. Its tracks are concentric circles. Each track is divided into a number of sectors. Its disk controller is somewhat more expensive than that required for reading. The advantages of WORM are its high capacity, longer life and better reliability.

The Erasable optical disk is read/write optical memory. The disk contents can be erased and new data can be rewritten to it. It is also used as secondary memory of computer. The tracks are concentric circle. The coating of an erasable optical disk is done by a magnetic material, which does not lost its magnetic properties at the room temperature. The reading and writing operations are performed using magneto-optical system. In which a laser beam is employed together with a magnetic field to read/write operations.

Working mechanism of Optical disks in case of CD vs. DVD

As it has been discussed above that an optical disc is made mainly of polycarbonate (a plastic) see the Fig. 6.4 given below. The data is stored on a layer inside the polycarbonate. A metal layer reflects the laser light back to a sensor. And to read the data on a disk, laser light shines through the polycarbonate and hits the data layer. How the laser light is reflected or absorbed is read as a 1 or a 0 by the computer.

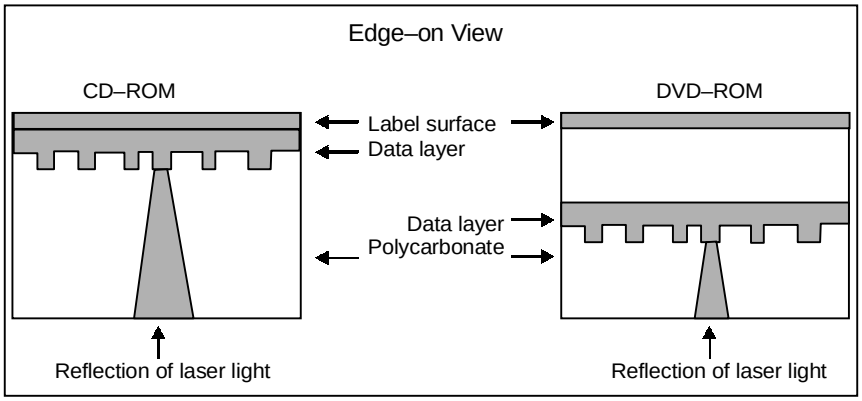


Fig. 6.4: Optical Disks (CD vs. DVD)

In a CD, the data layer is near the top of the disk, the label side. In a DVD the data layer is in the middle of the disk. A DVD can actually have data in two layers. It can access the data from 1 side or from both sides. This is how a double-sided, double-layered DVD can hold 4 times the data that a single-sided, single-layered DVD can. The CDs and DVDs that are commercially produced are of the Write Once Read Many (WORM) variety. They can't be changed once they are created.

Other Storage Devices—Flash Drives

Pen Drives

Also known as USB Flash Drive, USB Thumb Drive, Flash Drives. A thumb drive is portable memory storage. It is rewritable and holds its memory without a power supply, unlike RAM. Thumb drives will fit into any USB port on a computer. They will also "hot swap," which means a user can plug the drive into a computer and will not have to restart it to access the thumb drive. The drives are small, about the size of a human thumb hence, their name and are very stable memory storage devices. The thumb drive is available in storage sizes of up to 8 gigabytes (starting from 128MB, 256MB, 512MB, 1GB, 2GB, 4GB, 8GB). They are stable, versatile, durable and portable data storage devices. As such they are ideal for almost any computer user who wants safe, long-term storage for a low price. USB flash drives may have different design, different capacity and different price and some USB flash drives feature add-on functions such as MP3 players. But they do share some other characteristics:

USB flash drives are lightweight. Most USB flash drives are as light as a car key.

USB flash drives are small. Can be kept in your or attached with key chain.

USB flash drives carry large capacity of data, up to 8GB USB flash drives.

USB flash drives are helpful to store personal information without saving them in computer hard drive in case of sharing of a computer with other peoples at work place.

Tape Drives

A device, like a tape recorder, that reads data from and writes it onto a tape. Tape drives have data capacities of anywhere from a few hundred kilobytes to several gigabytes of information without having to spend large sums of money on disks. Their transfer speeds also vary considerably. Fast tape drives can transfer as much as 20MB (megabytes) per second. Tape Drives software is generally easy to use and can usually be ran without supervision. While Tape Drives are cost efficient and easy to use one major disadvantage. Tape Drives have the speed which they backup and recover information. Tape drives are a sequential access device, which means to read any data on the Tape Drive; the Tape Drive must read all preceding data. Tape drives are available in various design and shape like 8mm tape drive similar to what are used in camcorder with the transfer rate up to 6M/ Sec. Other is DLT (Digital Linear Tape) drive that is a robust and durable medium. The DLT segments the tape into parallel horizontal tracks and records data by streaming the tape across a single stationary head. Some other examples are DAT (Digital Audio Tape), QIC Standard. The disadvantage of tape drives is that they are *sequential-access* devices, which means that to read any particular block of data, it requires to read all the preceding blocks. This makes them much too slow for general-purpose storage operations. However, they are the least expensive media for making backups.

Zip Drives

Zip disks are high capacity(up to 100MB), removable, magnetic disks. ZIP disks are similar to floppy disks, except that they are much faster, and have a much greater capacity.

While floppy disks typically hold 1.44 megabytes, ZIP disks are available in two sizes, namely 100 megabytes and 250 megabytes. ZIP drives should not be confused with the super-floppy, a 120 megabyte floppy drive which also handles traditional 1.44 megabyte floppies. ZIP drives are available as internal or external units, using one of three interfaces:

- Small Computer Standard Interface (SCSI): Interface is the fastest, most sophisticated, most expandable, and most expensive interface. The SCSI interface is used by all types of computers from PC's to RISC workstations to minicomputers, to connect all types of peripherals such as disk drives, tape drives, scanners, and so on. SCSI ZIP drives may be internal or external, assuming your host adapter has an external connector.
- Integrated Drive Electronics (IDE): Interface is a low-cost disk drive interface used by many desktop PC's. Most IDE devices are strictly internal.
- The parallel port interface is popular for portable external devices such as external ZIP drives and scanners, because virtually every computer has a standard parallel port (usually used for printers). This makes things easy for people to transfer data between multiple computers by toting around their ZIP drive.

Zip disks can be used to store, backup, and move basic office application files, digital music, presentations, digital photos, digital video, etc. On the other hand, in spite of Iomega's claims that this drives "meet high capacity storage needs" for PC users, these products belong to the mobile storage rather than to the back-up category.

6.1.3 Network Protection Devices

System and network security is the term used to describe the methods and tools employed to prevent unauthorized and malicious access or modification of data on a system or during data transmission over network. Network security is not just for big corporations and government organizations only. The new breed of viruses, worms, and deceptive software that can infect computer or allow malicious hackers to unauthorized use of computers from any type of network interconnects. A bigger question arises what to protect on the network? The protection involves the following:

- Intrusion prevention.
- Intrusion detection.
- Web filtering.
- E-mail security.
- Security management.
- Integrated security appliance.
- Vulnerability assessment.

Network protection devices are used to preemptively protect computer network from viruses, worms and other Internet attacks. Intrusion detection and prevention, firewalls, vulnerability assessment, integrated security appliances, web filtering, mail security and a

centralized management system, all work to maximize your network uptime and minimize the need for active administrator involvement.

Firewall used for all these provides only one entry point to your network. And if the modems are allowed to answer incoming calls, can provide an easy means for an attacker to sneak around the firewall. Just as castles weren't built with moats only in the front, then network needs to be protected at all of its entry points.

Secure Modems; Dial-Back Systems: If modem access is to be provided, this should be guarded carefully. The *terminal server*, or network device that provides dial-up access to your network needs to be actively administered, and its logs need to be examined for strange behaviour. Its password need to be strong. Accounts that are not actively used should be disabled. In short, it is the easiest way to get into your network from remote: guard it carefully. There are some remote access systems that have the feature of a two-part procedure to establish a connection. The first part is the remote user dialing into the system, and providing the correct user-Id and password. The system will then drop the connection, and call the authenticated user back at a known telephone number. Once the remote user's system answers that call, the connection is established, and the user is on the network. This works well for folks working at home, but can be problematic for users wishing to dial in from hotel rooms and such when on business trips. Other possibilities include one-time password schemes, where the user enters his user-Id, and is presented with a "challenge," a string of between six and eight numbers.

Crypto-Capable Routers: A feature that is being built into some routers is the ability to session encryption between specified routers. Because traffic travelling across the Internet can be seen by people in the middle, who have the resources (and time) to snoop around, these are advantageous for providing connectivity between two sites, such that there can be secure routes.

Virtual Private Networks

Given the ubiquity of the Internet, and the considerable expense in private leased lines, many organizations have been building *VPNs* (Virtual Private Networks). Traditionally, for an organization to provide connectivity between a main office and a satellite one, an expensive data line had to be leased in order to provide direct connectivity between the two offices. Now, a solution that is often more economical is to provide both offices connectivity to the Internet. Then, using the Internet as the medium, the two offices can communicate. The danger in doing this, of course, is that there is no privacy on this channel, and it's difficult to provide the other office access to "internal" resources without providing those resources to everyone on the Internet. VPNs provide the ability for two offices to communicate with each other in such a way that it looks like they're directly connected over a private leased line. The session between them, although going over the Internet, is private (because the link is encrypted), and the link is convenient, because each can see each other's internal resources without showing them off to the entire world.

Wireless Network Protection

In case of wireless network, it requires to take an additional security steps when wireless access point is set up first. Wireless networks are protected by something called Wired Equivalent Privacy (WEP) encryption. There are two steps to enabling WEP:

Step-1 is the configuring the wireless access point: The wireless access point is the device that is probably connected to cable or DSL modem. Instructions for configuration will vary slightly for wireless access points from different manufacturers.

Step-2 is the configuring the wireless network adapter: The wireless network adapter is either plugged into computer, or that is built-in to computer. In case of an older wireless network adapter, it requires check with the manufacturer to find out which WEP key lengths it supports.

6.1.4 Surge Protectors

A surge is defined as a voltage increase that lasts for as little as three nanoseconds (one nanosecond is one billionth of a second), and significant damage can be done in that miniscule amount of time, if the voltage surge is strong enough. A spike, which lasts for only one or two nanoseconds, can also do its share of damage, especially when several spikes occur repeatedly over an extended period. Voltage surges and spikes occur for a number of reasons. Perhaps the most common is the sudden jump in voltage that occurs when high-power appliances such as refrigerators and air conditioners first start up. The appliances need quite a bit of electrical energy to activate compressors, and that sudden and sharp increase in flow through the lines will be felt by the electronics. A surge protector is necessary to protect electronics against "dirty" electricity. Electrical power has a standard voltage for most residential uses of 120 volts to 240 volts, and it remains relatively steady. But when that power makes a sharp and brief jump for any of a variety of reasons, the resulting sudden alteration in voltage can seriously damage delicate circuits.

Electricity is your computer's lifeblood. Power anomalies and surges also pose a big threat to computer equipment. But the power line that supplies your computer with electricity also carries the seeds of your computer's destruction. Surges that are brief pulses of high voltage they sneak down the power line and, in an instant, can destroy the circuits inside computer. The way to protect your computer from lightning and power surges is to use a good surge protector. A power strip, which is a simple strip of outlets, is not necessarily a surge protector. A surge protector may look like a power strip, but it has built-in protection against power surges. Surge protectors are simple to install, maintenance free and have become much more affordable. Surge protection units are available that offer four to six protected outlets in one protection "center," which makes it easy and convenient to protect not only the computer but the printer, fax, external modem, scanner and other home office components. Many of these units also offer surge protection for one or more phone lines. A good surge protector should offer four features.

- The surge protector should cover lightning strikes. Some do not.

- The surge protector should offer insurance to cover the loss of properly attached equipment.
- For a regular modem, get a surge protector with an R-11 telephone jack where it can be hooked up with telephone line.
- With a cable modem, use a surge protector which will also accommodate the television/Internet cable.

The performance of surge protectors is rated three ways that are clamping voltage, response time and energy absorption. The first, clamping voltage, tells what level of voltage surge has to occur before the surge protector activates and diverts the excess voltage to ground. With this rating, the lower the voltage number is the better the surge protector will perform. It takes less of a surge to activate. For good protection, especially for computers, a protector with a clamping voltage of less than 400 volts will be preferred. Response time is the amount of time it takes for the surge protector to respond to the surge. Obviously, a fast response time is important, so look for a unit that will respond in one nanosecond or less. Surge protectors are not made to last forever, so the third rating, energy absorption, indicates how much energy the unit will absorb before it fails. For this rating, look for a unit rated at 300 joules or better, up to around 600 joules for even better performance.

A surge protection strip or center is also well-suited for home entertainment components like TV, VCR, stereo, etc. While not as delicate as computers, providing good surge protection will certainly help extend the useful life of any of these components. Entertainment center surge protectors may also contain protection for a phone line and a cable TV line, and typically cost a little less than the ones designed for computer protection. Example of some most commonly used surge protectors are ISP3 Inline Surge Protector with audible alarm, ISP 4 (Inline Surge Protector), ISP 5-perfect protection, ISP6 (Inline Surge Protector) 'cloverleaf'.

UPS (Uninterruptible Power Supply)

A UPS (Uninterruptible Power Supply) is basically a battery back-up system to maintain power in the event of a power outage for computer. UPS provides power for a short time (usually 10 or 15 minutes) to the computer or other critical hardware when its primary power source is unavailable. A UPS keeps a computer running for several minutes after a power outage, enabling you to save data that is in RAM and shutdown the computer gracefully. Power spikes, sags, and outages can not only cause lose of unsaved work and precious data, they can also damage valuable hardware and network infrastructure.

It acts as a surge suppressor, filtering line noise and providing protection against spikes. But, in the event of a power outage it keeps your computer up and running, sounding an alarm and allowing you to close any running programs and shutdown your computer safely. There are various common power problems that UPS units are used to correct. They are as follows:

Power failure: Total loss of utility power, causes electrical equipment to stop working.

Voltage sag: Transient (short term) under-voltage that causes flickering of lights.

Voltage spike: Transient (short term) over-voltage i.e., spike or peak that causes wear or acute damage to electronic equipment.

Under-voltage (brownout): Low line voltage for an extended period of time that causes overheating in motors.

Over-voltage: Increased voltage for an extended period of time that causes light bulbs to fail.

Line noise: Distortions superimposed on the power waveform that causes electro-magnetic interference.

Frequency variation: Deviation from the nominal frequency (50 or 60 Hz) that causes motors to increase or decrease speed and line-driven clocks and timing devices to gain or lose time.

Switching transient: Instantaneous undervoltage (notch) in the range of nanoseconds. May cause erratic behaviour in some equipment, memory loss, data error, data loss and component stress.

Harmonic distortion: Multiples of power frequency superimposed on the power waveform that causes excess heating in wiring and fuses.

There are two basic types of UPS systems available in the market one is on-line UPS systems. And other one is off-line UPS systems (also known as standby power systems. An on-line UPS always powers the load from its own internal energy supply, which is in turn continuously charged by the input power. An SPS monitors the power line and switches to battery power as soon as it detects a problem. The switch to battery, however, can require several milliseconds, during which time the computer is not receiving any power. Standby Power Systems are sometimes called Line-interactive UPSs. An on-line UPS avoids these momentary power lapses by constantly providing power from its own inverter, even when the power line is functioning properly. In general, on-line UPSs are much more expensive than SPSs. In a standby (off-line) system the load is powered directly by the input power and the backup power circuitry is only invoked when the utility power fails.

Most UPS below 1 kVA are of the standby variety which are cheaper, though inferior to on-line systems which have no delay between a power failure and backup power being supplied. A true 'uninterruptible' system is a double-conversion system. In a double-conversion system alternating current (AC) comes from the power grid, goes to the battery (direct current or DC), then is converted back to AC power. Most systems sold for the general market, however, are of the "standby" type where the output power only draws from the battery, if the AC power fails or weakens. For large power units, Dynamic Uninterruptible Power Supply are sometimes used. A synchronous motor/alternator is connected on the mains via a choke. Energy is stored in a flywheel. When the mains fails, an Eddy-current regulation maintains the power on the load. DUPS are sometimes combined or integrated with a diesel-genset. In recent years, Fuel cell UPS have been developed that uses hydrogen and a fuel cell as a power source potentially providing long run times in a small space. A fuel cell replaces the batteries as energy storage used in all UPS design.

6.1.5 RAID Technology

RAID is also known as redundant array of independent disks or often incorrectly known as redundant array of inexpensive disks. RAID is a system of using multiple hard drives for sharing or replicating data among the drives. Depending on the version chosen, the benefit of RAID is one or more of increased data integrity, fault-tolerance, throughput or capacity compared to single drives. In its original implementations its key advantage is the ability to combine multiple low-cost devices using older technology into an array that offeres greater capacity, reliability, or speed, or a combination of these things, than affordably available in a single device using the newest technology.

At the very simplest level, RAID combines multiple hard drives into one single logical unit. Thus, instead of seeing several different hard drives, the operating system sees only one. RAID is typically used on server computers, and is usually implemented with identically-sized disk drives. With decreases in hard drive prices and wider availability of RAID options built into motherboard chipsets. RAID is also being found and offered as an option in more advanced end user computers. This is especially true in computers dedicated to storage-intensive tasks, such as video and audio editing.

The RAID specification suggests a number of prototype "RAID levels", or combinations of disks. Each had theoretical advantages and disadvantages. Over the years, different implementations of the RAID concept have appeared. Most differ substantially from the original idealized RAID levels, but the numbered names have remained. The very definition of RAID has been argued over the years. The use of the term *redundant* leads many to split hairs over whether RAID 0 is a "real" RAID type. Similarly, the change from *inexpensive* to *independent* confuses many as to the intended purpose of RAID. There are even some single-disk implementations of the RAID concept. For the purpose of this article, we will say that any system which employs the basic RAID concepts to recombine physical disk space for purposes of reliability, capacity, or performance is a RAID system. There are number of different RAID levels:

Level 0—RAID (Striped Disk Array without fault tolerance)

Level 1—RAID (Mirroring and Duplexing)

Level 2—RAID (Error-Correcting Coding)

Level 3—RAID (Bit-Interleaved Parity)

Level 4—RAID (Dedicated Parity Drive)

Level 5—RAID (Block Interleaved Distributed Parity)

Level 6—RAID (Independent Data Disks with Double Parity)

Nested RAID levels: Many storage controllers allow RAID levels to be nested. That is, one RAID can use another as its basic element, instead of using physical disks.

Hardware and Software of RAID

RAID can be implemented either in dedicated hardware or custom software running on standard hardware. Additionally, there are hybrid RAIDs that are partly software and

partly hardware-based solutions. With a software implementation, the operating system manages the disks of the array through the normal drive controllers like IDE (Integrated Drive Electronics)/ATA (Advanced Technology Attachment), SCSI (Small Computer System Interface) and Fibre Channel or any other. With present CPU speeds, software RAID can be faster than hardware RAID, though at the cost of using CPU power which might be best used for other tasks. One major exception is where the hardware implementation of RAID incorporates a battery backed-up write cache and an application, like a database server. In this case, the hardware RAID implementation flushes the write cache to a secure storage to preserve data at a known point if there is a crash. The hardware approach is faster and limited instead by RAM speeds, the amount of cache and how fast it can flush the cache to disk. For this reason, battery-backed caching disk controllers are often recommended for high transaction rate database servers. In the same situation, the software solution is limited to no more flushes than the number of rotations or seeks per second of the drives. Another disadvantage of a pure software RAID is that, depending on the disk that fails and the boot arrangements in use, the computer may not be able to be rebooted until the array has been rebuilt.

A hardware implementation of RAID requires (at a minimum) a special-purpose RAID controller. On a desktop system, this may be a PCI (Peripheral Component Interconnect) expansion card, or might be a capability built in to the motherboard. In larger RAIDs, the controller and disks are usually housed in an external multi-bay enclosure. The disks may be IDE, ATA, SATA, SCSI, Fibre Channel, or any combination thereof. The controller links to the host computer(s) with one or more high-speed SCSI, Fibre Channel or ISCSI (Internet SCSI) connections, either directly, or through a fabric, or is accessed as network attached storage. This controller handles the management of the disks, and performs parity {In computing and telecommunication, a parity bit is a binary digit that takes on the value zero or one to satisfy a constraint on the overall parity of a binary number. The *parity scheme* in use must be specified as even or odd (also called *even parity* and *odd parity*, respectively). Parity is even if there are an even number of '1' bits, and odd otherwise} calculations (needed for many RAID levels). This option tends to provide better performance, and makes operating system support easier. Hardware implementations also typically support hot swapping, allowing failed drives to be replaced while the system is running. In rare cases hardware controllers have become faulty, which can result in data loss. Because of this drawback, software RAID is a slightly more reliable and safer option.

Hybrid RAIDs have become very popular with the introduction of very cheap *hardware RAID controllers*. The hardware is just a normal disk controller that has no RAID features, but there is a boot-time set-up application that allows users to set up RAIDs that are controlled via the BIOS(Basic input output systems) . When any modern operating systems are used, they will need specialized RAID drivers that will make the array look like a single block device. Since these controllers actually do all the calculations in software, not hardware, they are often called "fakeraids". Unlike software RAID, these "fakeraids" typically cannot span multiple controllers.

Both hardware and software versions may support the use of a *hot spare* (A hot spare is a disk or group of disk used to automatically or manually, depending on the Hot spare policy, replace a failing disk in a RAID), a preinstalled drive which is used to immediately (and almost always automatically) replace a failed drive. This cuts down the time period in which a second failure can take out the array. Some software RAID systems allow building arrays from partitions instead of whole disks. Unlike Matrix RAID they are not limited to just RAID 0 and RAID 1 and not all partitions have to be RAID.

Reliability Factors of RAID

There are some important factors affecting the relaibility of RAID configuration like failure rate of disk, mean time of data loss and mean time of recovery.

Failure rate: A failure rate is the average frequency with which something fails. Failure rate can be defined as " The total number of failures within an item population, divided by the total time expended by that population, during a particular measurement interval under stated conditions. (MacDiarmid, et al.)" The meantime to failure of a given RAID may be lower or higher than those of its constituent hard drives, depending on what type of RAID is employed.

Mean Time to Data Loss (MTTDL): In this context, the meantime to elapse before a loss of user data in a given array, usually measured in hours.

Mean Time to Recovery (MTTR): Meantime to recovery is the average time that a device will take to recover from a non-terminal failure. Examples of such devices range from self-resetting fuses (where the MTTR would be very short, probably seconds), up to whole systems which have to be replaced. In arrays that include redundancy for reliability, this is the time following a failure to restore an array to its normal failure-tolerant mode of operation. This includes time to replace a failed disk mechanism as well as time to rebuild the array (i.e., to replicate data for redundancy).

6.1.6 Server Specific Jargon

In the client/server environment servers are also computers like other workstations with some configurational differences where processing speed is measured megahertz (MHz), hard disk capacity measured in gigabytes (GB), data transfer rates measured in milliseconds (MS); apart from these, there are some server specific jargon that are useful to know like EDC Memory, Memory Cache, Rack Mounting, Power Protection, RAID and Symmetrical Multiprocessing.

EDC Memory

Error Detection and Correction (EDC) such type of memory is configured at the hardware level with special circuitry that verifies RAM output and resends output whenever the memory errors occur. This type of memory is used to boost overall reliability of servers and tending to become standard equipment.

Memory Cache

Memory cache sets aside a portion of the server RAM to store the most frequently used network instructions so that these instructions can be accessed as soon as possible. While the server is in operation cache storage is being constantly updated. While network processing, less frequently accessed instructions are pushed out of cache, replaced by instructions that are accessed more frequently. The larger the size of the memory cache, the more instructions the server can keep on hand for fast access.

Rack Mounting

A rack mount server usually refers to multiple servers stacked on top of one another in a single cabinet to save space. In case of very large client/server a system that requires more than a single file server rack mount can be used, where the system is complex and highly centralized.

Power Protection

Power supply is a unit that distributes electricity within the server. A RDS redundant power supply is a backup power supply that takes over in the event that the main power supply fails. This feature is different from an UPS (uninterruptible power supply) an external device that provides continuous electrical power to the server, usually for a short time, in the event of an electrical power failure. Details of UPS has already discussed in Section 6.1.4, i.e. surge protectors, RDS keeps the network running indefinitely, as long as electricity is being fed to it on other hand UPS keeps the network running just long enough after a power failure to store and protect data before shuting down. A line conditioner that is another form of power protectors can be used to monitor the electrical current and compensates for extreme fluctuations (i.e. *spikes,* burst of too much voltage or *brownouts,* sudden drop in voltage).

Symmetrical Multiprocessing

Symmetrical Multiprocessing (SMP) technology is used to integrate the power of more than one central processor into a single file server, along with necessary additional hardware and software to divide processing chores between them. There is a drastic effect on the server speed by using multiple processors, although it is not as simple as doubling speed with two processors or tripling speed with three. SMP involves additional processing overhead to manage the distribution of processing among those multiple processors. In case of big client/server systems where a large scale networks is involved the features of SMP are used.

6.2 SOFTWARE REQUIREMENTS

6.2.1 Client OS

The client always provides presentation services, all the user Input and Output are presented at client workstation. Software to support specific functions like field edits, context-sensitive help, navigation, training, personal data storage, and manipulation frequently get executes on the client workstation. All these functions use the GUI and windowing functionality. Additional business logic for calculations, selection, and analysis can reside on the client workstation. A client workstation uses a local operating system to host both basic services and the network operating system interfaces. This operating system may be the same or different from that of the server. Numbers of OS are installed depending upon the application and user requirement running on Client/Server environment. There are various OS are in use as a client platform like DOS, Windows 3.1, OS/2, UNIX, Windows NT (New Technology), AIX and Mc systems 7. The client workstation frequently provides personal productivity functions, such as word processing, which use only the hardware and software resident right on the workstation. When the client workstation is connected to a LAN, it has access to the services provided by the network operating system (NOS) in addition to those provided by the client workstation. The workstation may load software and save word-processed documents from a server and therefore use the file server functions provided through the NOS. It also can print to a remote printer through the NOS. The client workstation may be used as a terminal to access applications resident on a host minicomputer or mainframe processor. This enables the single workstation to replace the terminal, as well as provide client workstation functionality.

6.2.2 Server OS

Servers provide the platform for application, database, and communication services also the server provides and controls shared access to server resources. Applications on a server must be isolated from each other so that an error in one cannot damage another. Preemptive multitasking ensures that no single task can take overall the resources of the server and prevent other tasks from providing service. There must be a means of defining the relative priority of the tasks on the server. These requirements are specific to the Client/Server implementation and not to the file server implementation. Because file servers execute only the single task of file service, they can operate in a more limited operating environment without the need for application isolation and preemptive multitasking.

The server is a multiuser computer. There is no special hardware requirement that turns a computer into a server. The hardware platform should be selected based on application demands and economics. There is no pre-eminent hardware technology for the server. The primary characteristic of the server is its support for multiple simultaneous client requests for service. Therefore, the server must provide multitasking support and

shared memory services. Servers for Client/Server applications work best when they are configured with an operating system that supports shared memory, application isolation, and preemptive multitasking. High-end Intel, RISC (including Sun SPARC, IBM/Motorola PowerPC, HP PA RISC, SGI MIPS, and DEC Alpha), IBM System/370, and DEC VAX processors are all candidates for the server platform. The server is responsible for managing the server-requester interface so that an individual client request response is synchronized and directed back only to the client requester. This implies both security when authorizing access to a service and integrity of the response to the request. Some of the operating system dominating the server word nowadays are NetWare, Windows NT, OS/2, MVS, VMS, and UNIX.

NetWare

In 2003, Novell announced the successor product to NetWare (Open Enterprise Server OES). Later on completes the separation of the services traditionally associated with NetWare like directory services, file, and printer from the platform underlying the delivery of those services. OES is essentially a set of applications (eDirectory, NetWare Core Protocol services, iPrint, etc.) that can run a top either a Linux or a NetWare kernel platform. Also known as self-contained operating system so does not requires separate operating system to run.

OS/2

The last released version was 4.0 in 1996. Early versions found their way into embedded systems and still, as of mid-2003, run inside many of the world's automated teller machines. Like Unix, OS/2 was built to be preemptively multitasking and would not run on a machine without an MMU (early versions simulated an MMU using the 286's memory segmentation). Unlike Unix, OS/2 was never built to be a multiuser system. Process-spawning was relatively cheap, but IPC was difficult and brittle. Networking was initially focused on LAN protocols, but a TCP/IP stack was added in later versions. There were no programs analogous to Unix service daemons, so OS/2 never handled multi-function networking very well. OS/2 had both a CLI and GUI. Most of the positive legendary around OS/2 was about the Workplace Shell (WPS), the OS/2 desktop. The combination of Novell with an OS/2 database and application servers can provide the necessary environment for a production-quality Client/Server implementation.

Windows NT

Windows NT (New Technology) is Microsoft's operating system released in september 1993, for high-end personal and server use. Microsoft staked its unique position with a server operating system. Microsoft's previous development of OS/2 with IBM did not create the single standard UNIX alternative that was hoped for. NT provides the preemptive multitasking services required for a functional server. It provides excellent support for Windows clients and incorporates the necessary storage protection services required for a reliable server operating system.

NT has file attributes in some of its file system types. They are used in a restricted way, to implement access-control lists on some file systems, and do not affect development style very much. It also has a record-type distinction, between text and binary files, that produces occasional annoyances (both NT and OS/2 inherited this misfeature from DOS).

NT systems on the Internet are notoriously vulnerable to worms, viruses, defacements, and cracks of all kinds. There are many reasons for this, some more fundamental than others. The most fundamental is that NT's internal boundaries are extremely porous. Because Windows does not handle library versioning properly, it suffers from a chronic configuration problem called "DLL hell", in which installing new programs can randomly upgrade (or even downgrade!) the libraries on which existing programs depend. This applies to the vendor-supplied system libraries as well as to application-specific ones: it is not uncommon for an application to ship with specific versions of system libraries, and break silently when it does not have them. On the bright side, NT provides sufficient facilities to host Cygwin, which is a compatibility layer implementing Unix at both the utilities and the API level, with remarkably few compromises. Cygwin permits C programs to make use of both the Unix and the native APIs, and is the first thing many Unix hackers install on such Windows systems as they are compelled by circumstances to make use of. The intended audience for the NT operating systems is primarily nontechnical end users, implying a very low tolerance for interface complexity. It is used in both client and server roles. Early in its history Microsoft relied on third-party development to supply applications. They originally published full documentation for the Windows APIs, and kept the price of development tools low. But over time, and as competitors collapsed, Microsoft's strategy shifted to favor in-house development, they began hiding APIs from the outside world, and development tools grew more expensive.

MVS

MVS (Multiple Virtual Storage) is IBM's flagship operating system for its mainframe computers as a platform for large applications. MVS is the only one OS that could be considered older than Unix. It is also the least influenced by Unix concepts and technology, and represents the strongest design contrast with Unix. The unifying idea of MVS is that all work is batch; the system is designed to make the most efficient possible use of the machine for batch processing of huge amounts of data, with minimal concessions to interaction with human users. MVS uses the machine MMU; processes have separate address spaces. Interprocess communication is supported only through shared memory. There are facilities for threading (which MVS calls "subtasking"), but they are lightly used, mainly because the facility is only easily accessible from programs written in assembler. Instead, the typical batch application is a short series of heavyweight program invocations glued together by JCL (Job Control Language) which provides scripting, though in a notoriously difficult and inflexible way. Programs in a job communicate through temporary files; filters and the like are nearly impossible to do in a usable manner. The intended role of MVS has always been in the back office. Like VMS and Unix itself, MVS predates the server/client distinction. Interface complexity for back-office users is not only tolerated

but expected, in the name of making the computer spend fewer expensive resources on interfaces and more on the work it's there to get done.

VMS

OpenVMS is a multi-user, multiprocessing virtual memory-based operating system (OS) designed for use in time sharing, batch processing, real time (process priorities can be set higher than OS kernel jobs) and transaction processing. It offers high system availability through clustering, or the ability to distribute the system over multiple physical machines. This allows the system to be "disaster-tolerant" against natural disasters that may disable individual data-processing facilities. VMS also includes a process priority system that allows for real-time process to run unhindered, while user processes get temporary priority "boosts" if necessary Open VMS commercialized many features that are now considered standard requirements for any high-end server operating system. OpenVMS commercialized many features that are now considered standard requirements for any high-end server operating system. These include Integrated computer networking, a distributed file system , Integrated database features and layered databases including relational database, Support for multiple computer programming languages, Hardware partitioning of multiprocessors, High level of security. Enterprise class environments typically select and use OpenVMS for various purposes including as a mail server, network services, manufacturing or transportation control and monitoring, critical applications and databases, and particularly environments where system uptime and data access is critical.

UNIX

Unix operating system developed in 1969 by a group of AT&T employees at Bell Labs including Ken Thompson, Dennis Ritchie and Douglas McIlroy. During the late 1970s and early 1980s, Unix's influence in academic circles led to large-scale adoption of Unix by commercial startups, the most notable of which is Sun Microsystems. Today, in addition to certified Unix systems, Unix-like operating systems such as Linux and BSD derivatives are commonly encountered. Sometimes, "traditional Unix" may be used to describe a Unix or an operating system that has the characteristics of either Version 7 Unix or UNIX System V.

Unix operating systems are widely used in both servers and workstations. The Unix environment and the Client/Server program model were essential elements in the development of the Internet and the reshaping of computing as centered in networks rather than in individual computers. Unix was designed to be portable, multi-tasking and multi-user in a time-sharing configuration. Unix systems are characterized by various concepts like the use of plain text for storing data, a hierarchical file system, treating devices and certain types of inter-process communication (IPC) as files and the use of a large number of small programs that can be strung together through a command line interpreter using pipes, as opposed to using a single monolithic program that includes all of the same functionality. Unix operating system consists of many of these utilities along with the master control program, the kernel. The kernel provides services to start and stop programs, handle the file system and other common "low level" tasks that most programs share, and, perhaps most importantly, schedules

access to hardware to avoid conflicts if two programs try to access the same resource or device simultaneously. To mediate such access, the kernel was given special rights on the system, leading to the division between *user-space* and *kernel-space.*

6.2.3 Network OS

A Network Operating System (NOS) is a systen software that controls a network and its message (e.g., packet) traffic and queues, controls access by multiple users to network resources such as files, and provides for certain administrative functions, including security. Also includes special functions for connecting computers and devices into a local-area network (LAN) or Inter-networking. A Network Operating System (NOS) is an operating system that has been specifically written to keep networks running at optimal performance with a native structure for use in a network environment. Some of the important features of Network Operating System includes:

- Provide file, print, web services, back-up and replication services.
- Provide basic operating system features such as support for processors, protocols, automatic hardware detection and support multi-processing of applications.
- Security features such as authentication, authorization, logon restrictions and access control.
- Provide name and directory services.
- User management and support for logon and logoff, remote access, system management, administration and auditing tools with graphic interfaces.
- Support Internetworking such as routing and WAN ports.

Some of the components that an NOS usually has built in that a normal operating system might not have are built in NIC (network interface card) support, file sharing, server log on, drive mapping, and native protocol support. Most operating systems can support all of these components with add-on either by the original manufacture of the operating system or from a third party vendor. Some of the operating system dominating the networking OS are Novell NetWare, LAN Manager, IBM LAN Server, Banyan VINES etc.

Novell NetWare

NetWare is a network operating system developed by Novell, Inc. The latest version of NetWare is v6.5 Support Pack 7, which is identical to OES 2, NetWare Kernel. It initially used cooperative multitasking to run various services on a PC, and the network protocols were based on the archetypal Xerox XNS stack. NetWare has been superseded by Open Enterprise Server (OES). With Novell NetWare, disk space was shared in the form of NetWare volumes, comparable to DOS volumes. Clients running MS-DOS would run a special Terminate and Stay Resident (TSR) program that allowed them to *map* a local drive letter to a NetWare volume. Clients had to log in to a server in order to be allowed to map volumes, and access could be restricted according to the login name. Similarly, they could connect to the shared printers on the dedicated server, and print as if the printer was connected locally.

Novell had introduced limited TCP/IP support in NetWare v3.x (circa 1992) and v4.x (circa 1995), consisting mainly of FTP services and UNIX-style LPR/LPD printing (available in NetWare v3.x), and a Novell-developed webserver (in NetWare v4.x). Native TCP/IP support for the client file and print services normally associated with NetWare was introduced in NetWare v5.0. Most network protocols in use at the time NetWare was developed didn't trust the network to deliver messages. A typical client file read would work something like this:

- Client sends read request to server.
- Server acknowledges request.
- Client acknowledges acknowledgement.
- Server sends requested data to client.
- Client acknowledges data.
- Server acknowledges acknowledgement.

In contrast, NCP was based on the idea that networks worked perfectly most of the time, so the reply to a request served as the acknowledgement. Here is an example of a client read request using this model:

- Client sends read request to server.
- Server sends requested data to client.

All requests contained a sequence number, so if the client didn't receive a response within an appropriate amount of time it would re-send the request with the same sequence number. If the server had already processed the request it would re-send the cached response, if it had not yet had time to process the request it would send a 'positive acknowledgement' which meant, "I received your request but I haven't gotten to it yet so don't bug me." The bottom line to this 'trust the network' approach was a 2/3 reduction in network traffic and the associated latency. In 4.x and earlier versions, NetWare did not support preemption, virtual memory, graphical user interfaces etc. Processes and services running under the NetWare OS were expected to be cooperative, that is to process a request and return control to the OS in a timely fashion. On the down side, this trust of application processes to manage themselves could lead to a misbehaving application bringing down the server.

LAN Manager

LAN Manager is a network operating system developed by Microsoft developed in cooperation with 3Com (Computers, Communication and Compatibility) that runs as a server application under OS/2. It supports DOS, Windows and OS/2 clients. LAN Manager provides client support for DOS, Windows, Windows NT, OS/2, and Mac System 7. Server support extends to NetWare, AppleTalk, UNIX, Windows NT, and OS/2. Client workstations can access data from both NetWare and LAN Manager Servers at the same time. LAN Manager supports NetBIOS and Named Pipes LAN communications between clients and OS/2 servers. Redirection services are provided to map files and printers from remote workstations for client use. LAN Manager was superseded by Windows NT Server, and many parts of LAN Manager are used in Windows NT and 2000.

IBM LAN Server

A network operating system developed by IBM that runs as a server application under OS/2 and supports DOS, Windows and OS/2 clients. Originally based on LAN Manager when OS/2 was jointly developed by IBM and Microsoft, starting with LAN Server 3.0, it runs only under IBM's version of OS/2. Short term LAN Server refers to the IBM OS/2 LAN Server product. There were also LAN Server products for other operating systems, notably AIX (now called Fast Connect) and OS/400. LAN server is a file server in a network. LAN Server provides disk mirroring, CID capability and Network Transport Services/2 (NTS/2) for concurrent access to NetWare servers. Options are LAN Server for the Macintosh for Mac client access and System Performance/2 (SP/2), a series of network management utilities. LAN Server, are the standard products for use in Client/Server implementations using OS/2 as the server operating system. LAN Manager/X is the standard product for Client/Server implementations using UNIX System V as the server operating system.

Banyan VINES

Banyan VINES (Virtual Integrated Network Service) is developed during 1980. Banyan VINES is a computer network operating system and set of computer network protocols, it used to talk to client machines on the network. In other words Banyan VINES is a network operating system with a UNIX kernel that allows clients operating systems such as DOS, OS/2, Windows, and those for Macintosh systems to share information and resources with each other and with host computing systems. VINES provide full UNIX NFS (Network File System) support in its core services and the Transmission Control Protocol/Internet Protocol (TCP/IP) for transport, it also includes Banyan's StreetTalk Directory Services, one of the first viable directory services to appear in a network operating system.

VINES ran on a low-level protocol known as VIP (VINES Internetwork Protocol) essentially identical to the lower layers of XNS), addresses consisted of a 32-bit address and a 16-bit subnet, which mapped onto the 48-bit Ethernet address in order to route to machines. This meant that, like other XNS-based systems, VINES could only support a two-level internet. However, a set of routing algorithms set VINES apart from other XNS systems at this level. The key differentiator, ARP (Address Resolution Protocol), allowed VINES clients to automatically set up their own network addresses. When a client first booted up it broadcast a request on the subnet asking for servers, which would respond with suggested addresses. The client would use the first to respond, although the servers could hand off better routing instructions to the client if the network changed. The overall concept very much resembled AppleTalk's AARP system, with the exception that VINES required at least one server, whereas AARP functioned completely headlessly. Like AARP, VINES required an inherently chatty network, sending updates about the status of clients to other servers on the internetwork. At the topmost layer, VINES provided the standard file and print services, as well as the unique StreetTalk, likely the first truly practical globally consistent name service for an entire internetwork. Using a globally distributed, partially replicated database, StreetTalk could meld multiple widely separated networks into a single network that allowed seamless resource sharing. It accomplished

this through its rigidly hierarchical naming scheme; entries in the directory always had the form *item@group@organization*. This applied to user accounts as well as to resources like printers and file servers. VINES client software ran on most PC-based operating systems, including MS-DOS and earlier versions of Microsoft Windows. It was fairly light weight on the client, and hence remained in use during the later half of the 1990s, when many machines not up to the task of running other networking stacks then in widespread use. This occurred on the server side as well, as VINES generally offered good performance even from mediocre hardware.

6.3 COMMUNICATION INTERFACE TECHNOLOGY

For the data communication to be taking place on a network, four basic elements are involved there:

Sender: the device that creates and transmits the data.

Message: the data to be sent. It could be a spreadsheet, database, or document, converted to digital form.

Medium: the physical material that connects the devices and carries the data from the sender to the receiver. The medium may consist of an electrical wire or airwaves.

Receiver: the destination device for the data.

To communicate with other devices, a sending device must know and follow the rules for sending data to receiving devices on the network. These rules for communication between devices are called *protocols*. Numerous standards have been developed to provide common foundations for data transmission. The International Standards Organization (ISO) has divided the required communication functions into seven levels to form the Open Systems Interconnections (OSI) model. Each layer in the OSI model specifies a group of functions and associated protocols used at that level in the source device to communicate with the corresponding level in the destination device.

Connectivity and interoperability between the client and the server are achieved through a combination of physical cables and devices and software that implements communication protocols. To communicate on a network the following components are required:

- A network interface card (NIC) or network adapter.
- Software driver.
- Communication protocol stack.

Computer networks may be implemented using a variety of protocol stack architectures, computer buses or combinations of media and protocol layers, incorporating one or more of among the LAN Cabling, WAN, Ethernet, IEEE NIC, Token Ring, Ethernet and FDDI.

6.3.1 Network Interface Card

The physical connection from the computer to the network is made by putting a network interface card (NIC) inside the computer and connecting it to the shared cable. A network

interface card is a device that physically connects each computer to a network. This card controls the flow of information between the network and the computer. The circuit board needed to provide network access to a computer or other device, such as a printer. Network interface cards, or NICs, mediate between the computer and the physical media, such as cabling, over which transmissions travel. NIC is an adapter card that is installed in the controller that allows it to connect to a network (for example, Ethernet and Token Ring etc. The card contains both the hardware to accommodate the cables and the software to use the network's protocols. The NIC is also called a network adapter card.

6.3.2 LAN Cabling

LAN is data communication network, which connects many computers or client workstations and permits exchange of data and information among them within a localized area (2 to 5 Km). Where all connected devices share transmission media (cable) and also each connection device can work either stand alone or in the network. Each device connected in the network can communicate with any other device with a very high data transmission rate that is of 1Mbps to 100Mbps. Due to rapid change in technology, design and commercial applications for the LANs the number of approaches has emerged like High speed wireless LAN fast Ethernet. At the result, in many applications the volume of data handled over the LAN has been increased. For example in case of centralized server farms there is need for higher speed LAN. There is a need for client system to be able to draw huge amount of data from multiple centralized servers.

6.3.3 WAN

WAN (Wide area network) is a data communications network that covers a large geographical area such as cities, states or countries. WAN technologies generally function at the lower three layers of the OSI reference model, the physical layer, the data-link layer, and the network layer. WAN consists of a number of interconnected switching nodes via telephone line, satellite or microwaves links. A transmission form any one device is routed through internal nodes to the specific destination device. In WAN two computing device are not connected directly, a network of 'switching nodes' provides a transfer path and the process of transferring data block from one node to another is called data switching. Further this switching technique utilizes the routing technology for data transfer. Whereas the routing is responsible for searching a path between source and destination nodes. Earlier WAN have been implemented using circuit or packet switching technology, but now frame relay, ATM and wireless networks are dominating the technology.

WANs use numerous types of devices that are specific to WAN environments. WAN switches, access servers, bridge, gateway, repeater, brouter, modems, CSU/DSUs and ISDN terminal adapters. Other devices found in WAN environments that are used in WAN implementations include routers, ATM switches, and multiplexers.

6.3.4 ATM

Asynchronous Transfer Mode (ATM) is a connection-oriented technology, in which a logical connection is established between the two end points before the actual data exchange begins. ATM has proved very successful in the WAN scenario and numerous telecommunication providers have implemented ATM in their wide-area network cores ATM is a cell relay, packet switching network and data link layer protocol which encodes data traffic into small (53 bytes; 48 bytes of data and 5 bytes of header information) fixed-sized cells. ATM provides data link layer services that run over Layer 1 links. This differs from other technologies based on packet-switched networks (such as the Internet Protocol or Ethernet), in which variable sized *packets* (known as *frames* when referencing layer 2) are used. The motivation for the use of small data *cells* was the reduction of jitter (delay variance, in this case) in the multiplexing of data streams; reduction of this (and also end-to-end round-trip delays) is particularly important when carrying voice traffic. An ATM network is designed to be able to transfer many different types of traffic simultaneously, including real time flows such as video, voice and bursty TCP flows. ATM services are categorised into mainly two categories one is Real-Time Services and other one is Non-real-Time Services which are used by an end system to identify the type of service required. RTS concerns the delay and the variability of delay, referred to as jitter, that the application can tolerate. Real time applications typically involve a flow of information to a user that is intended to reduce that flow at a source. Constant Bit Rate services are the simplest real time services. CBR are used by the applications that requires a fixed data rate that is continuously available during the connections lifetime and a relatively tight upper bound on transfer delay. CBR applications are used mostly in video conferencing, interaction audio and audio/video retrieval and distribution. Real time variable bit rate (rtVB) are another real-time services that allows the network more flexibility than CBR. The network is able to statistically multiplex a number of connections over the same dedicated capacity and still provide the required service to each connection.

6.3.5 Ethernet

Ethernet is a family of frame-based computer networking technologies for Local Area Networks (LANs) that is also based on the idea of computers communicating over a shared coaxial cable acting as a broadcast transmission medium. The name comes from the physical concept of the ether. It defines a number of wiring and signaling standards for the physical layer, through means of network access. The communication methods used shows some similarities to radio systems, although there are fundamental differences, such as the fact that it is much easier to detect collisions in a cable broadcast system than a radio broadcast. The coaxial cable was replaced with point-to-point links connected by hubs and/or switches to reduce installation costs, increase reliability, and enable point-to-point management and troubleshooting. StarLAN was the first step in the evolution of Ethernet from a coaxial cable bus to a hub-managed, twisted-pair network.

Eathernet is most widely used LAN technology to get connected PCs and workstations more than 84% world wide due to its protocol that has following characteristics:

- Is easy to understand, implement, manage, and maintain.
- Allows low-cost network implementations.
- Provides extensive topological flexibility for network installation.
- Guarantees successful interconnection and operation of standards.
- Compliant products, regardless of manufacturer.

Ethernet LANs consist of network nodes and interconnecting media. The network nodes fall into two major classes:

- Data Terminal Equipment (DTE)—Devices that are either the source or the destination of data frames. DTEs are typically devices such as PCs, workstations, file servers, or print servers that, as a group, are all often referred to as end stations.
- Data Communication Equipment (DCE)—Intermediate network devices that receive and forward frames across the network. DCEs may be either stand alone devices such as repeaters, network switches, and routers, or communications interface units such as interface cards and modems.

6.3.6 Token Ring

Token-Ring was developed and promoted by IBM in the early 1980s and standardized as IEEE 802.5. Physically, a token ring network is wired as a star, with 'hubs' and arms out to each station and the loop going out-and-back through each. Stations on a token ring LAN are logically organized in a ring topology with data being transmitted sequentially from one ring station to the next with a control token circulating around the ring controlling access. Token ring is a local area network protocol which resides at the Data Link Layer (DLL) of the OSI model. It uses a special three-byte frame called a token that travels around the ring. Token ring frames travel completely around the loop.

Token-passing networks move a small frame, called a token, around the network. Possession of the token grants the right to transmit. If a node receiving the token has no information to send, it passes the token to the next end station. Each station can hold the token for a maximum period of time. If a station possessing the token does have information to transmit, it seizes the token, alters 1 bit of the token (which turns the token into a start-of-frame sequence), appends the information that it wants to transmit, and sends this information to the next station on the ring. While the information frame is circling the ring, no token is on the network (unless the ring supports early token release), which means that other stations wanting to transmit must wait. Therefore, collisions cannot occur in Token Ring networks. Token ring networks had significantly superior performance and reliability compared to early shared-media implementations of Ethernet (IEEE 802.3), and were widely adopted as a higher-performance alternative to the shared-media Ethernet.

6.3.7 FDDI

FDDI (Fiber Distributed Data Interface), as a product of American National Standards Institute X3T9.5 (now X3T12), conforms to the Open Systems Interconnection (OSI) model of functional layering of LANs using other protocols.

FDDI provides a standard for data transmission in a local area network that can extend in range up to 200 kilometers. In addition to covering large geographical areas, FDDI local area networks can support thousands of users. As a standard underlying medium, it uses optical fiber (though it can use copper cable, in which case one can refer to CDDI). A FDDI network contains two token rings (dual-ring architecture) with traffic on each ring flowing in opposite directions (called counter-rotating). The dual rings consist of a primary and a secondary ring. During normal operation, the primary ring is used for data transmission, and the secondary ring remains idle. Secondary ring also provides possible backup in case the primary ring fails. The primary ring offers up to 100 Mbit/s capacity. When a network has no requirement for the secondary ring to do backup, it can also carry data, extending capacity to 200 Mbit/s. The single ring can extend the maximum distance; a dual ring can extend 100 km. FDDI has a larger maximum-frame size than standard 100 Mbit/s ethernet, allowing better throughput. The primary purpose of the dual rings is to provide superior reliability and robustness.

6.3.8 TCP/IP

The Internet protocol suite is the set of communications protocols that implement the protocol stack on which the Internet and most commercial networks run. It has also been referred to as the TCP/IP protocol suite, which is named after two of the most important protocols in it: the Transmission Control Protocol (TCP) and the Internet Protocol (IP). TCP/IP is referred as protocol suite because it contains many different protocols and therefore many different ways for computers to talk to each other. TCP/IP is not the only protocol suite, although TCP/IP has gained wide acceptance and is commonly used. TCP/IP also defines conventions by connecting different networks, and routing traffic through routers, bridges, and other types of connections. The TCP/IP suite is result of a Defence Advanced Research Projects Agency (DARPA) research project about network connectivity, and its availability has made it the most commonly installed network software.

6.3.9 SNMP

The Simple Network Management Protocol (SNMP) forms part of the internet protocol suite as defined by the Internet Engineering Task Force (IETF). SNMP is used in network management systems to monitor network-attached devices for conditions that warrant administrative attention. It consists of a set of standards for network management, including an Application Layer protocol, a database schema, and a set of data objects.

SNMP exposes management data in the form of variables on the managed systems, which describe the system configuration. These variables can then be queried (and sometimes set) by managing applications. In typical SNMP usage, there are a number of

systems to be managed, and one or more systems managing them. A software component called an *agent* runs on each managed system and reports information via SNMP to the managing systems. An SNMP-managed network consists of three basic key components:

- Managed devices
- Agents
- Network-Management Systems (NMSs)

A managed device is a network node that contains an SNMP agent and that resides on a managed network. Managed devices collect and store management information and make this information available to NMSs using SNMP. Managed devices, sometimes called network elements, can be any type of device including, but not limited to, routers and access servers, switches and bridges, hubs, IP telephones, computer hosts, or printers.

An agent is a network-management software module that resides in a managed device. An agent has local knowledge of management information and translates that information into a form compatible with SNMP.

An NMS executes applications that monitor and control managed devices. NMSs provide the bulk of the processing and memory resources required for network management. One or more NMSs may exist on any managed network.

6.3.10 NFS

Network File System (NFS) is a network file system protocol originally developed by Sun Microsystems in 1984, allowing a user on a client computer to access files over a network as easily as if the network devices were attached to its local disks. NFS, like many other protocols, builds on the Open Network Computing Remote Procedure Call (ONC RPC) system. Assuming a Unix-style scenario in which one machine (the client) requires access data, stored on another machine (the NFS server).

The server implements NFS daemon processes (running by default as NFSD) in order to make its data generically available to clients. The server administrator determines what to make available, exporting the names and parameters of directories (typically using the/etc./exports configuration file and the exports command).

The server security-administration ensures that it can recognize and approve validated clients. The server network configuration ensures that appropriate clients can negotiate with it through any firewall system. The client machine requests access to exported data, typically by issuing a mount command. If all goes well, users on the client machine can then view and interact with mounted file systems on the server within the parameters permitted.

6.3.11 SMTP

Simple Mail Transfer Protocol (SMTP) is the standard for e-mail transmissions across the Internet developed during 1970's. SMTP is a relatively simple, text-based protocol, in which one or more recipients of a message are specified (and in most cases verified to exist) and

then the message text is transferred. It is a Client/Server protocol, whereby a client transmits an e-mail message to a server. Either an end-user's e-mail client, a.k.a. MUA (Mail User Agent), or a relaying server's MTA (Mail Transfer Agents) can act as an *SMTP client*. An email client knows the *outgoing mail* SMTP server from its configuration. A relaying server typically determines which SMTP server to connect to by looking up the MX (Mail eXchange) DNS record for each recipient's domain name (the part of the e-mail address to the right of the at (@) sign). Conformant MTAs (not all) fall back to a simple A record in the case of no MX. Some current mail transfer agents will also use SRV records, a more general form of MX, though these are not widely adopted. (Relaying servers can also be configured to use a smart host. SMTP is a "push" protocol that does not allow one to "pull" messages from a remote server on demand. To do this a mail client must use POP3 or IMAP. Another SMTP server can trigger a delivery in SMTP using ETRN.

An e-mail client requires the name or the IP address of an SMTP server as part of its configuration. The server will deliver messages on behalf of the user. This setting allows for various policies and network designs. End users connected to the Internet can use the services of an e-mail provider that is not necessarily the same as their connection provider. Network topology, or the location of a client within a network or outside of a network, is no longer a limiting factor for e-mail submission or delivery. Modern SMTP servers typically use a client's credentials (authentication) rather than a client's location (IP address), to determine whether it is eligible to relay e-mail.

One of the limitations of the original SMTP is that it has no facility for authentication of senders. Therefore, the SMTP-AUTH extension was defined. However, the impracticalities of widespread SMTP-AUTH implementation and management means that E-mail spamming is not and cannot be addressed by it.

EXERCISE 6

1. In a typical Client/Server under Network environment, explain the following in details:
   (*a*) What are the Server requirements?
   (*b*) What are the H/W requirements?
   (*c*) What are the Client requirements?
   (*d*) What are the Network requirements?
   (*e*) What do you mean by a thin client network?
   (*f*) List some advantages of Thin Client Network system.
2. Microsoft Windows NT Server provides various network services to support specific requirements of the users on the network. All network services impact the capacity of a network. Some of the services are:

(*a*) Net Logon
(*b*) Computer Browser
(*c*) DHCP
(*d*) Internet Explorer
(*e*) Workstation
(*f*) Server

Explain the above part in brief in a Client/Server environment.

3. In Client/Server architecture in a network environment, explain the following phenomena examples:–
   (*a*) UPS
   (*b*) Surge Protector
   (*c*) Optical Disk
   (*d*) CDDI
4. Explain the functions and features of Network Operating System.
5. Explain the working principal of following:
   (*a*) CDROM
   (*b*) WORM
   (*c*) Mirror disk
   (*d*) Tape optical disk
   (*e*) UNIX Workstation
   (*f*) Notebooks
6. In design a network operating system; discuss the relative advantages and disadvantages of using a single server and multiple servers for implementing a service.
7. Write short notes on the following:
   (*a*) X-Terminals
   (*b*) RAID Array Disk
   (*c*) FDDI
   (*d*) Power Protection Devices
   (*e*) Network Interface Cards
   (*f*) Network Operating System
   (*g*) Fax Print Services
8. Explain how microkernels can be used to organize an operating system in a Client/Server fashion.
9. What are different client hardware and software for end users? Explain them.

7

Training and Testing

7.1 INTRODUCTION

In addition to being an important factor in the successful implementation of a Client/Server system, training makes employees aware of security concerns. It also educates employees for needed behavioural changes to comply with internal controls. Additionally, it provides employees with the basic knowledge to operate well in a new system environment.

User training for a Client/Server system is complicated by the interface of multiple front-end systems with servers. The user-friendly design provides users with a variety of options for their applications. As front-end applications vary, so do the training and technical support needs. Training of programming and support personnel is also complicated because the underlying system is more complex than traditional systems.

To teach the fundamental technologies involved in a modern Client/Server system in an easy manner, so that one can understand the business requirements, design issues, scalability and security issues in an enterprise system. Teaching style is explanation of concepts, supported by hands on examples. Apart from this required training, it needs training for system administrator, database administrator training, end user training. Existing training delivery vehicles are being revolutionized using new information and telecommunication systems. Classroom delivery can now take place in a synchronous mode across the entire planet. Trainees can participate in learning activities at any time and location using Internet and satellite technologies. These vehicles are breaking the boundaries of space and time, offering unlimited learning possibilities to organizations and those who work for them.

The effectiveness of training in Client/Server computer application development depends on the combination of instructor-led and team-based learning methods. Ten persons familiar with the course content generated 90 statements and each subsequently, completed

a similarity sort of the statements. The concept map showed eleven clusters ranging from contextual supports and general methodology on the top and left, to more specific Client/ Server concepts on the right and bottom. The cluster describing the actual coding and testing of Client/Server applications are very important for training. The Fig. 7.1 given below shows the concept map for training in Client/Server computer application development.

One consistent pattern that emerged is that the highest achievement tended to be evidenced in clusters most related to the way the course was taught, a combination of instructor-led and team-based learning. The clusters related to communication, teaming and coaching consistently placed in the top five across all administrations. One implication may be that the training methods used may have been more salient than the content.

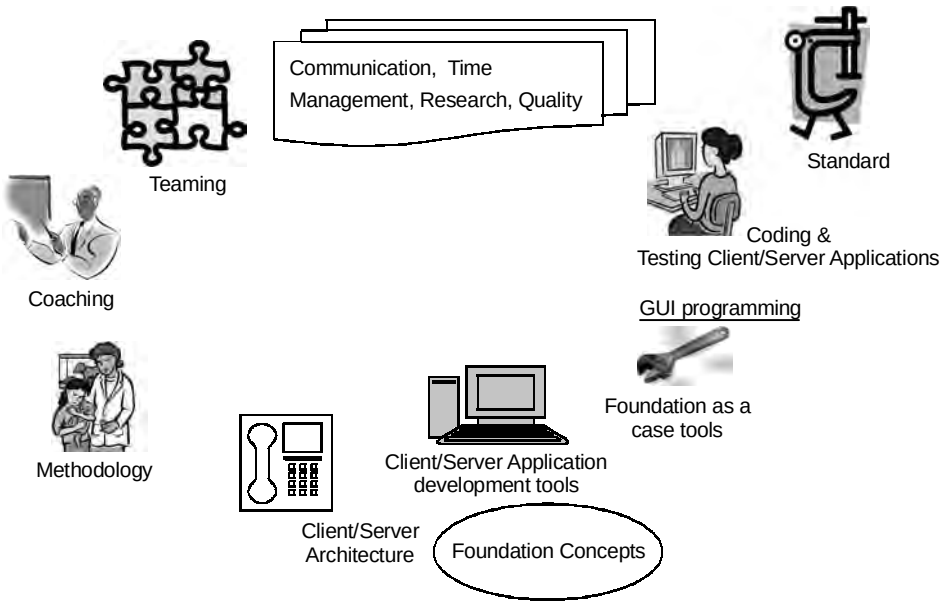


Fig. 7.1: Concept Map for Training in Client/Server Computer Application Development

7.2 TECHNOLOGY BEHIND TRAINING DELIVERY

7.2.1 Traditional Classroom

Most of you are familiar with the traditional face-to-face method of training and learning, given in a classroom or seminar. It is the oldest method for delivering training, and it can still get the job done effectively. In classical training delivery, learners and the instructor are present at the same time. Therefore, classroom training is defined as a synchronous training delivery vehicle. Face-to-face experience provides the trainer and participant with

immediate feedback. It enables participants to discuss and share ideas with others in the classroom or seminar. The trainer presents materials, manages and guides discussion; responsibly ensuring that learning is constructive and positive. Traditional classroom delivery is thus termed teacher centric. The time commitment needed for traditional classroom delivery is increasingly considered a drawback in this kind of approach. With downsizing, rightsizing and outsourcing, there are fewer people with time to sit in a classroom. In addition, staff scheduling is often a nightmare when attendance is low or if the training must be scheduled according to employee commitments.

7.2.2 On-the-Job Training (OTJ)

On-the-job training is also a classical training delivery approach, dating back to the middle ages, when apprenticeship was dominant as a learning form, and little formal training existed. Often one of the most popular training delivery vehicles, OTJ training is frequently confused with informal office discussions, round table exchanges and brainstorming sessions. It is sometimes difficult to precisely define what makes up OTJ training. Frequently your coach, mentor, or trainer has a checklist of items which they must demonstrate to the learner, and validate learner's comprehension. Alternatively, informal styles consist of asking a learner to repeat an activity until the coach, mentor, or trainer is satisfied with the learner's performance.

7.2.3 Video Conferencing

There are many conferencing and virtual meeting tools, but most can be placed in one of two distinct categories: Conference room video conferencing, where two or more groups exchange information using one or two-way visual (image) and two-way audio (voice) transmission. Typically, wired conference rooms are voice activated. The person speaking dominates the audio lines. Students can see the instructor, and the instructor can often view the class groupings, sometimes with the capacity to focus in on the person speaking. Computer conferencing, where exchange information using one way visual (image) and two way (voice) transmission is employed. If all computers are equipped with cameras, peer to peer exchange—such as instructor to student and student to student allows both image and voice exchange. Streaming media technology will increasingly be used internally at companies and in business-to-business ventures and that will drive up corporate spending on the technology. This training delivery vehicle offers a live, immediate and synchronous experience, presenting a good discussion format in real time, with equipment that is relatively easy to operate.

7.2.4 Collaborative Tools

A host of conferencing tools can streamline and enrich a virtual business meeting experience, including, but not limited to, digital video, program and document sharing, and whiteboarding. These tools help the staff to find new ways to learn and collaborate online in real time. Digital whiteboards employ a board, a pen and an eraser to store every word,

line and color on the computer. Using an Electronic Projection System (EPS), trainer can share this information electronically over the network with learners, creating an electronic flipchart in real time. In collaborative systems, learners can add comments to the flipchart that are visible to all session participants.

7.2.5 Virtual Groups and Event Calls

Computer technology allows training by using virtual groups in a cheap, accessible and easy-to-use fashion. Set up chat rooms where learning groups can discuss, debate and share information after training content has been distributed. Chat groups lack the face-to-face element and can be asynchronous if members are not asked to be simultaneously available. Voice intonation and body language cannot guide the learners. Virtual groups should be managed, focused and kept small, since superficial treatment of materials can be frustrating to learners. Some computers may be slower than others, so delays in communication should be expected. Other virtual groups include virtual office sites, where members of a company can interact using a computer, Web cam and modem to conduct meetings from any geographical location in the world. Event call training involves use of the telephone only rather than the computer. Training materials are often sent to event call locations in advance, and are delivered one-way by the instructor. Participants in many locations, connected via telephone conference, can ask questions. Again, telephone event calling offers no face-to-face element. However, it serves as an inexpensive, quick way to diffuse a uniform message to staff in different locations.

7.2.6 E-Learning

E-learning is the hot word on the block in training and investment circles. The term is elusive, and means something a little different to everyone. In terms of learning delivery approach e-learning is asynchronous: the learner does not receive instruction in the presence of an instructor or other students. The learner can repeat a lesson as many times as he needs, extracting the parts of the course he requires without wasting time going through material he has already mastered. Learners can proceed through an electronic program at their own pace, stopping and starting as desired. E-learning can be designed to offer different levels of complexity, targeting a wider training audience and customizing training accordingly. E-learning encompasses Computer Based Training (CBT), using a computer in combination with Compact Disks with Read-Only Memory (CD-ROMs), Digital Video Disks (DVDs) or browser driven, Web-Based Training (WBT). E-learning can be networked or single user based. E-learning vehicles depend on the technology available and bandwidth capacity. Lower bandwidth means that fewer graphic, animation, sound and video elements are possible.

7.2.7 Web-based Training

Web-based training is browser driven. For this reason, it is more accessible to many, but expensive for some because the Internet access is charged by the minute.

Accessibility to e-learning "is not currently as integral as an employee manual, a college syllabus or a 9th grade math textbook. Most industry observers and education practitioners believe that one day soon, it will be. Web-based content can be easily changed or updated so that learners receive the most recent version. When training is complete, feedback in the form of test or quiz results can be given online and stored in databases or Learning Management Systems. Instructor feedback and follow-up can take the form of online chat rooms or e-mail. Universal accessibility to the Web might require using limited bandwidth, which results in slower performance for sound and images. Avoid long downloading delays, since this can be a source of frustration for users. This module is a sample of low bandwidth, interactive, Web-based solution.

7.2.8 Learning Management Systems (LMS)

Learning Management Systems have been developed to record learner progress in computer and Web-based training. Some systems incorporate both asynchronous and synchronous training. Features generally include coordinating course registration, scheduling, tracking, assessment and testing learners while reporting to managers. Many systems interface with human resource development and enterprise wide systems. Learning Management Systems track and manage the learning process for each user. Some contain course and knowledge management modules. These are termed learning course management systems. There are approximately 600 currently on the market. Learning Management Systems can consume much of an infrastructure budget, so careful consideration should be given before selecting one. Another negative impact of implementing an LMS is that learners may feel policed. This may reduce learners' willingness to use this type of e-learning product.

7.2.9 Electronic Performance Support Systems (EPSS)

An Electronic Performance Support System provides task specific information, training, coaching, and monitoring to enhance job performance. The key to good EPSS tools is their simplicity and accuracy. An EPSS can be in the form of help files, glossary items, and task tools available on the Internet, or in print. EPSSs are concise, efficient to use, and provide clarification on tasks and procedures. An EPSS is part online help, part online tutorial, part database, part application program, and part expert system. In short, an EPSS is an integrated version of a lot of products that technical communicators produce. It is "an electronic system that directly supports a worker's performance when, how, and where the support is needed."

7.3 TO WHOM TRAINING IS REQUIRED?

7.3.1 System Administrator Training

System administrator is the person in Client/Server environment, who understands the availability of resources desired by client. He must understand the level of system

performance and ease of use their users requirement. System Administrator concentrates on tasks that are independent of the applications running on adaptive server; he or she is likely to be the person with the best overview of all the applications. System administrator is responsible for managing server, client and as well as about all the applications running in the environment. Some of the important system administrator's tasks on that management must concentrate to prove sufficient training to system administrator.

1. Setting up and managing client server database, managing and monitoring the use of disk space, memory, and connections, backing up and restoring databases server, integration with back-end databases.
2. Setting up and managing user accounts, granting roles and permissions to Adaptive Server users and managing remote access.
3. Working with various control panels and hosting automation software and also day-to-day management of the equipment.
4. Diagnosing system problems along with fault management and performance management.

7.3.2 DBA Training

Client/Server environment consists of centralized or distributed data, so database administrator requires additional responsibilities. He must perform skilled operations with the help of technical staff while operating the application running on client server model. The additional complexity of the new environment requires some new training for database administration staff in that case design of Client/Server environment plays an important role in effecting the performance due to location of data. Here, Database Administrator (DBA) is an experienced senior member(s) of the computing staff who plan and co-ordinate the development and daily maintenance of the entire database environment. He has an extensive working knowledge of Client/Server environment. Usually the role of DBA is shared between 2 or 3 such employees for security purposes. A typical DBA's duties include:

- Installing and configuring the DBMS.
- Assisting in the implementation of information systems.
- Monitoring the performance of the database and tuning the DBMS for optimal performance.
- Ensuring data integrity is maintained and appropriate backups are made.
- Resource usage monitoring.
- Setting standards for system documentation.
- Facilitating end-users with required database facilities.
- Overseeing new database developments, database re-organisation.
- Maintaining an acceptable level of technical performance for database utilities.
- Educating the organization in the use, capabilities and availability of the database.

In other words, DBA prime basic duties are to manage administrative procedures like installation and configuration, backup, recovery, security administration and performance tuning that covers application, database, Client/Server, parallel, restructuring, crisis management, corruption repairs, long running and platform specific tuning.

7.3.3 Network Administrator Training

Network administrators training program specifically focuses on the design, installation, maintenance and management as well as implementation, and operating network services on LAN (Local-Area Network), WAN (Wide-Area Network), network segment, Internet, intranet of Client/Server system. The increasing decentralizes activities of network services makes coordinated network management difficult. To mange a network, let us see what the basic fundamentals are associated with a network management system. Basically, network management system is a collection of application, software, hardware and some tools for monitoring and controlling the system integration as shown in Fig. 7.2 given below. Then training of Network System Administrator requires training of following important areas like:

- Configuration management.
- Performance management.
- Accounting management.
- Security management.
- Fault management.

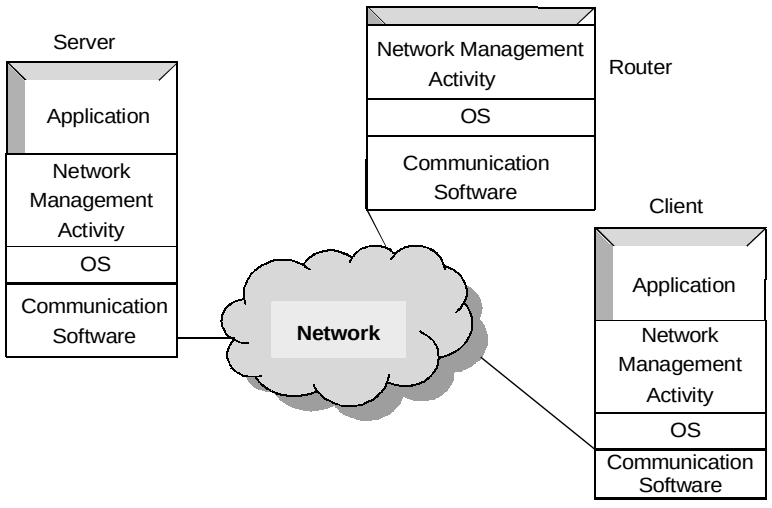


Fig. 7.2: Network System Elements

We can say, Network Systems Administrator are responsible for ensuring an organization's networks are used efficiently under the Client/Server environment. They

provide day-to-day administrative support, monitor systems and make adjustments as necessary, and trouble-shoot problems reported by client and automated monitoring systems. They also gather data regarding customer needs, and then evaluate their systems based on those needs. In addition, they may also be involved in the planning and implementation of network security systems.

7.3.4 End-User and Technical Staff Training

End user's are the user's of Client/Server environment those how are already having sufficient knowledge about application running on the system. They are not technical persons but having enough function knowledge of system. They need to train about some new standards and functionality and technology of the applications being implemented in the system.

Technological component constituting the Client/Server system must be completely known to the supporting staff. There must a interface between high level administration and technical staff so that database access and communication technologies can be used in maximum potential by any client. Technical staff must be trained to respect the technological knowledge of user who is already familiar with existing system. The technical staff is the middle level of user of client server applications, then their corporate experience counts while a major fault occurs with system.

7.3.5 GUI Applications Training

Most clients in Client/Server systems deliver system functionality using a Graphical User Interface (GUI). When testing complete systems, the tester must grapple with the additional functionality provided by the GUI. GUIs have become the established alternative to traditional forms-based user interfaces. GUIs are the assumed user interface for virtually all systems development using modern technologies. There are several reasons why GUIs have become so popular that includes:

- GUIs provide the standard look and feel of a client operating system.
- GUIs are so flexible that they can be used in most application areas.
- The GUI provides seamless integration of custom and package applications.
- The user has a choice of using the keyboard or a mouse device.
- The user has a more natural interface to applications: multiple windows can be visible simultaneously, so user understanding is improved.
- The user is in control: screens can be accessed in the sequence the user wants at will.
- The most obvious characteristic of GUI applications is the fact that the GUI allows multiple windows to be displayed at the same time. Displayed windows are 'owned' by applications and of course, there may be more than one application active at the same time.

Windows provide forms-like functionality with fields in which text or numeric data can be entered. But GUIs introduce additional objects such as radio buttons, scrolling lists,

check boxes and other graphics that may be displayed or directly manipulated. The GUI itself manages the simultaneous presentation of multiple applications and windows. Hidden windows in the same or different applications may be brought forward and used. There are few, if any, constraints on the order in which users access GUI windows so users are free to use the features of the system in the way they prefer, rather than the way the developers architected it. However, the sophistication and simplicity of a GUI hides the complexity from the user and where development frameworks are used, the programmers too. When trainers are presented with a GUI application to train, the hidden complexities become all too obvious. Consequently, training GUIs is made considerably more difficult. There are some key points that must be taken care while providing training to the various applications on Client/Server environment, in fact, these are the difficulties associated with GUI training.

- ***Event-driven nature:*** The event-driven nature of GUIs presents the first serious training difficulty. Because users may click on any pixel on the screen, there are many, many more possible user inputs that can occur. The user has an extremely wide choice of actions. At any point in the application, the users may click on any field or object within a window. They may bring another window in the same application to the front and access that. The window may be owned by another application. The user may choose to access an operating system component directly e.g., a system configuration control panel.
- ***Unsolicited events:*** Unsolicited events cause problems for trainers. A trivial example would be when a local printer goes off-line, and the operating system puts up a dialog box inviting the user to feed more paper into the printer. A more complicated situation arises where message-oriented middleware might dispatch a message (an event) to remind the client application to redraw a diagram on screen, or refresh a display of records from a database that has changed. Unsolicited events may occur at any time, so again, the number of different situations that the code must accommodate is extremely high. Training of unsolicited events is difficult.
- ***Hidden synchronization and dependencies:*** It is common for window objects to have some form of synchronization implemented. For example, if a check box is set to true, a text box intended to accept a numeric value elsewhere in the window may be made inactive or invisible. If a particular radio button is clicked, a different validation rule might be used for a data field elsewhere on the window. Synchronization between objects need not be restricted to object in the same window and its training must be done carefully.
- ***'Infinite' input domain:*** On any GUI application, the user has complete freedom to click with the mouse-pointing device anywhere on the window that has the focus. Although objects in windows have a default tab order, the user may choose to enter data values by clicking on an object and then entering data.

- ***Many ways in, many ways out:*** An obvious consequence of the event-driven nature of GUIs is that for most situations in the application, there may be 'many ways in' by which the user reached that point in the application. As many as possible, the ways must be known to the user.
- ***Window management:*** In a GUI environment, users take the standard features of window management and control for granted. These features include window movement, resizing, maximization, minimization and closure. These are usually implemented by standard buttons and keyboard commands available on every window. The trainer has control over which standard window controls are available, but although the operating system handles the window's behavior, the trainer must handle the impact on the application.

7.3.6 LAN/WAN Administration and Training Issues

For LAN administration there are various products available such as Network General Sniffer that enables administrator to monitor the network for capacity and problems without the need for detail knowledge of the applications. The biggest advantage of using such software is that they can be used to monitor LAN traffic, analyzing the data, and then to recommend actions based on data assessment. The software interpreter internal LAN message formats for LAN administrator to take action based on recommendations without the need for detailed knowledge of such message formats.

Before starting the training of any client/server system, the environment of system must be clearly known to administrator. Administrator must understand naming, security, help procedure etc., and able to implement them uniformly between applications and procedures. In case of large systems that are located in wide areas it requires administrator training as well as user training. Such training ensures that each of the installation operates in the same ways and also the support personal at remote can communicate with local administrator. All the software products should be installed on the entire client with uniform default setting and also the administrator should be an expert in the use of the product. Training document of product usage also revels that the administrator must understand, what the product requirements are? And arrange to have temporary and backup files created on volumes that can be cleaned up regularly.

WAN administrator must be trained in such a way that he can use and manage the remote management tools. Such tools enables administrator to remotely manage the LAN to WAN environment needed for many client/server applications. All the WAN network issues associated with remote terminal access to host system exist in the client/server to WAN access. Complexities arise when data is distributed to the remote LAN. The distributed application programs to remote servers present many of the same problems as do distributed databases. Then, the administrator must be trained in the software and in procedures to handle network definition, network management and remote backup and recovery. Due to wide impact of the WAN on communication issues, training developers in WAN issues becomes critical. And also due to availability of many optional configuration WAN's are

complex to understand and optimized. Then the training of WAN administrators to understand all of the options available to establish an optional topology becomes more expensive.

In client/server application administrator must be expert in the operating system used by clients and servers. The network used in client/server implementations frequently runs several operating systems. Such diversity of platforms changes the administrator to have expertise not only in the particular of a single operating system but also in the interaction of the various operating systems. While designing and planning for a new client/server application, the training requirements should be considered carefully before an organization establishes too many operating systems on the network. The cost and implications of training in this area must not be overlooked.

7.4 IMPACT OF TECHNOLOGY ON TRAINING

Client/Server Systems were initially developed as a cost-effective alternative to hosting business applications on Mainframes. Client/Server Systems offer many advantages over traditional systems such as low cost, increased performance and reliability due to a distributed design, and easier interoperability with other systems etc. Over the last decade, the second generation of Client/Server Systems has seen a major technological evolution. The earlier Client/Server Systems based on the two-tier database centric design have evolved to incorporate middleware technologies (for application logic) such as CORBA, COM+ and EJBs in an N-tier architecture. These technologies provide enhanced scalability and robustness to the business application. In the last few years, the Client/Server applications have been web-enabled for easier access and manageability. The integration of web and Client/Server technologies has given rise to important applications such as Data Warehousing, Online Analytical Processing (OLAP) and Data Mining. The necessities to integrate and exchange information with other business systems have led to the recent development of "Web Services" using XML and SOAP protocols. And of course, with web enabling, the security issues with Client/Server computing have become more important than ever. There are a number of factors driving the education and training markets to increase the use of technology for learning delivery:

Technical obstacles in adoption are falling: Network and system infrastructures, hardware access, and limited bandwidth are rapidly becoming non-factors.

Penetration of the Internet: The pervasiveness and familiarity of the Internet and its related technologies is the number one driver behind the growth of e-learning.

Market consolidation and one-stop shopping: Corporations, educational institutions and students are increasingly demanding more of their educational providers.

Traditional players looking to get on the scene: Many big industries and technology players are watching and waiting for market opportunities.

Knowledge is the competitive weapon of the 21st century: Knowledge is now the asset that will make or break a company. To remain competitive, corporations and individuals themselves–are expected to increase the amount spent on education to increase the value of their human capital.

7.4.1 Client/Server Administration and Management

Administration and management of Client/Server environments is an important and challenging area. Client/Server administration includes a range of activities: software distribution and version management, resource utilization monitoring, maintaining system security, reliability, and availability. Centralized mainframe environments are relatively easy to manage and are typically associated with high-level of security, data integrity, and good overall system availability. The present lack of administrative control over Client/Server environments is a major de-motivating factor for many organizations, who are considering migration from mainframe based systems. Personal Computer based networks are particularly difficult to administer and significant resources are needed to maintain Personal Computer environments in operation.

Client/Server administrators need to continuously monitor and pro-actively manage the system to ensure system availability. The key to effective Client/Server administration is fast identification of problem areas and fast failure recovery. Ideally, administrators should be able to anticipate critical situations using information derived by monitoring important resources. This allows intervention before the problems escalate and affect users of the system. Because of the complexity of distributed environments and the interaction between various system components, it is no longer possible to rely on traditional techniques, where the administrator interacts directly with the system using operating system commands. Automation of systems administration is an essential pre-requisite for the successful implementation of Client/Server systems.

7.5 CLIENT/SERVER TESTING TECHNOLOGY

7.5.1 Client/Server Software

Client/Server Software requires specific forms of testing to prevent or predict catastrophic errors. Servers go down, records lock, Input/Output errors and lost messages can really cut into the benefits of adopting this network technology. Testing addresses system performance and scalability by understanding how systems respond to increased workloads and what causes them to fail. Software testing is more than just review. It involves the dynamic analysis of the software being tested. It instructs the software to perform tasks and functions in a virtual environment. This examines compatibility, capability, efficiency, reliability, maintainability, and portability. A certain amount of faults will probably exist in any software. However, faults do not necessarily equal failures. Rather, they are areas of slight unpredictability that will not cause significant damage or shutdown. They are more

errors of semantics. Therefore, testing usually occurs until a company reaches an acceptable defect rate that doesn't affect the running of the program or at least won't until an updated version has been tested to correct the defects. Since Client/Server technology relies so heavily on application software and networking, testing is an important part of technology and product development. There are two distinct approaches when creating software tests. There is black box testing and white or glass box testing. Black box testing is also referred to as functional testing. It focuses on testing the internal machinations of whatever is being tested, in our case, a client or server program. When testing software, for example, black box tests focus on Input/Output. The testers know the input and predicted output, but they do not know how the program arrives at its conclusions. Code is not examined, only specifications are examined. Black box testing does not require special knowledge of specific languages from the tester. The tests are unbiased because designers and testers are independent of each other. They are primarily conducted from the user perspective to ensure usability. However, there are also some disadvantages to black box testing. The tests are difficult to design and can be redundant.

Also, many program paths go uncovered since it is realistically impossible to test all input streams. It would simply take too long. White box testing is also sometimes referred to as glass box testing. It is a form of structural testing that is also called clear box testing or open box testing. As expected, it is the opposite of black box testing. It focuses on the internal workings of the program and uses programming code to examine outputs. Furthermore, the tester must know what the program is supposed to do and how it's supposed to do it. Then, the tester can see if the program strays from its proposed goal. For software testing to be complete both functional/black and structural/white/glass box testing must be conducted.

7.5.2 Client/Server Testing Techniques

There are a variety of testing techniques that are particularly useful when testing client and server programs. Among them Risk Driven Testing and Performance Testing are most important.

(*a*) Risk Driven Testing: Risk driven testing is time sensitive, which is important in finding the most important bugs early on. It also helps because testing is never allocated enough time or resources. Companies want to get their products out as soon as possible. The prioritization of risks or potential errors is the engine behind risk driven testing. In risk driven testing the tester takes the system parts he/she wants to test, modules or functions, for example, and examines the categories of error impact and likelihood. Impact, the first category, examines what would happen in the event of a break-down. For example, would entire databases be wiped out or would the formatting just be a little off? Likelihood estimates the probability of this failure in the element being tested. Risk driven testing prioritizes the most catastrophic potential errors in the service of time efficiency.

(*b*) **Performance Testing:** Performance testing is another strategy for testing client and server programs. It is also called load testing or stress testing. Performance testing evaluates system components, such as software, around specific performance parameters, such as resource utilization, response time, and transaction rates. In order to performance test a client-server application, several key pieces of information must be known. For example, the average number of users working simultaneously on a system must be quantified, since performance testing most commonly tests performance under workload stress. Testers should also determine maximum or peak user performance or how the system operates under maximum workloads. Bandwidth is another necessary bit of information, as is most users' most frequent actions. Performance testing also validates and verifies other performance parameters such as reliability and scalability. Performance testing can establish that a product lives up to performance standards necessary for commercial release. It can compare two systems to determine which one performs better. Or they can use profilers to determine the program's behavior as it runs. This determines which parts of the program might cause the most trouble and it establishes thresholds of acceptable response times.

7.5.3 Testing Aspects

There are different types of software testing that focus on different aspects of IT architecture. **Three-testing** are particularly relevant to Client/Server applications. These are unit testing, integration testing, and system testing. A unit is the smallest testable component of a program. In object-oriented programming, which is increasingly influencing Client/Server applications. The smallest unit is a class. Modules are made up of units. Unit testing isolates small sections of a program (units) and tests the individual parts to prove they work correctly. They make strict demands on the piece of code they are testing. Unit testing documentation provides records of test cases that are designed to incorporate the characteristics that will make the unit successful. This documentation also contains positive and negative uses for the unit as well as what negative behaviors the unit will trap. However, unit testing won't catch all errors. It must be used with other testing techniques. It is only a phase of three-layer testing, of which unit testing is the first. Integration testing, sometimes called I&T (Integration and Testing), combines individual modules and tests them as a group. These test cases take modules that have been unit tested, they test this input with a test plan. The output is the integrated system, which is then ready for the final layer of testing, system testing. The purpose of integration testing is to verify functionality, performance, and reliability. There are different types of integration testing models. For example, the Big Bang model is a time saver by combining unit-tested modules to form an entire software program (or a significant part of one). This is the 'design entity' that will be tested for integration.

However, record of test case results is of the essence, otherwise further testing will be very complicated. Bottom up integrated testing tests all the low, user level modules, functions and procedures. Once these have been integrated and tested, the next level of modules can be integrated and tested. All modules at each level must be operating at the same level for this type of testing to be worthwhile. In object-oriented programming, of which client server applications increasingly are, classes are encapsulations of data attributes and functions. Classes require the integration of methods. Ultimately, integration testing reveals any inconsistencies within or between assemblages or the groupings of modules that are integrated through testing plans and outputs.

System testing is the final layer of software testing. It is conducted once the system has been integrated. Like integration testing, it falls within the category of black box testing. Its input is the integrated software elements that have passed integration testing and the integration of the software system with any hardware systems it may apply to. System testing detects inconsistencies between assemblages (thereby testing integration) and in the system as its own entity. System testing is the final testing front and therefore the most aggressive. It runs the system to the point of failure and is characterized as destructive testing. Here are some of the areas systems testing covers: usability, reliability, maintenance, recover, compatibility, and performance.

7.5.4 Measures of Completeness

In software testing, there are two measures of completeness, code coverage and path coverage. Code coverage is a white box testing technique to determine how much of a program's source code has been tested. There are several fronts on which code coverage is measured. For example, statement coverage determines whether each line of code has been executed and tested. Condition coverage checks the same for each evaluation point. Path coverage establishes whether every potential route through a segment of code has been executed and tested. Entry/Exit coverage executes and tests every possible call to a function and return of a response. Code coverage provides a final layer of testing because it searches for the errors that were missed by the other test cases. It determines what areas have not been executed and tested and creates new test cases to analyze these areas. In addition, it identifies redundant test cases that won't increase coverage.

7.6 TESTING CLIENT/SERVER APPLICATION

The structure of Client/Server Systems requires new approaches to testing them. A system can go wrong due to input/output errors, server down, records locked, lost messages and many more and then it requires to test the system response for these events. Performance and scalability testing is done to get valid results based on an insightful analysis of how systems respond to increasing load, and what makes them fail in various ways. With these information loads, testing is performed. Testing Client/Server applications requires some additional techniques to handle the new effects introduced by the Client/Server

architecture. Testing Client/Server Systems is entirely different; still the testing of software is there, see Fig. 7.3 given below.

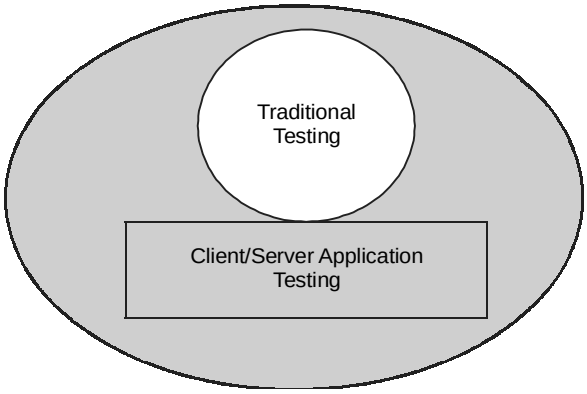


Fig. 7.3: Client/Server Testing

Such testing includes all the core testing techniques that are required to test *any* system, including systems that have a Client/Server design, *plus* the special techniques needed for Client/Server. Testing Client/Server applications is more challenging than testing traditional systems due to the following reasons:

- New kinds of things can go wrong: for example: Data and messages can get lost in the network.
- It's harder to set up, execute, and check the test cases: for example: Testing for proper responses to timeouts.
- Regression testing is harder to automate: for example: It's not easy to create an automated 'server is busy' condition.
- Predicting performance and scalability become critical: for example: It seems to work fine with 10 users. But what about with 1,000 users or 10,000 users?

Obviously, some new techniques are needed to test Client/Server systems. Most of these apply to distributed systems of all kinds, not just the special case of Client/Server architecture. The key to understanding how to test these systems is to understand exactly how and why each type of potential problem *arises*. With this insight, the testing solution will usually be obvious. For example, suppose Program A, (a client program, because it makes a request) asks Program B, a server program, to update some fields in a database.

Program B is on another computer. Program A expects Program B to report that either the operation was successfully completed or the operation was unsuccessful (for example, because the requested record was locked.). However, time passes and A hears nothing. What should A do? Depending on the speed and reliability of the network connecting A and B, there comes a time when A must conclude that something has probably gone wrong. Then, some possibilities are given as follow:

- The request got lost and it never reached B.
- B received the request, but is too busy to respond yet.
- B got the request, but crashed before it could begin processing it.
  B may or may not have been able to store the request before it crashed.
  If B did store the request, it might try to service it upon awakening.
- B started to process the request, but crashed while processing it.
- B finished processing the request (successfully or not), but crashed before it could report the result.
- B reported the result, but the result got lost and was never received by A.

There are more possibilities, but you get the idea. So what should A do?

Here the problem is that A can't tell the difference between any of the above cases (without taking some further action). Dealing with this problem involves complex schemes such as the Two Phases Commit. When to test the client program A, its needed to see whether it's robust enough to at least do something intelligent for each of the above scenarios. This example illustrates one new type of testing that have been done for Client/ Server systems. Testing the client program for correct behavior in the face of uncertainty. Now, let's look at the other new type of testing, this time on the server side with consideration of performance testing and scalability issues. There are some major reasons why Client/Server systems cause new effects:

(*i*) **Most of these systems are *event-driven*:** Basically, this means: "Nothing Happens Until Something Happens". Most program action is triggered by an event, such as the user hitting a key, some Input/Output being completed, or a clock timer expiring. The event is intercepted by an "event handler" or "interrupt handler" piece of code, which generates an internal message (or lots of messages) about what it detected. This means that it's harder to set up test cases than it is, say, to define a test case for a traditional system that prints a check. To set up a test case it requires to create events, or to simulate them. That is not always easy, especially, because it needed to generate these events when the system is in the proper state; but there are ways to do it.

(*ii*) **The systems never stop:** Many Client Server/Systems are set up to never stop running unless something goes really wrong. It is true for the servers, and in many cases, it is also true for the client machines. Traditional systems complete an action, such as printing a report, and then turn in for the night. When user restarts the program it is a whole fresh new day for it. In systems that don't stop (on purpose), things are different. Errors accumulate. As someone put it, sludge builds up on the walls of the operating system. So things like **memory leaks** that are wrong, but that probably would not affect the most traditional systems, will eventually bring non-stop systems down if they are not detected and corrected. One good way to minimize these effects is to use something called SOW and HILT variables in testing.

The system contains multiple computers and processors which can act (and fail) independently. Worse, they communicate with each other over less than perfectly reliable communication lines. These two factors are the root cause of the problem detailed above, where Program A makes a request of Program B, but gets no response.

EXERCISE 7

1. In Client/Server architecture, you are appointed as a system administrator. You have about 500 users and it is a mix of WinNT machine, Macintosh machine, and a very few DOS machine. Describe all the part given below with example, wherever necessary.
   (*a*) What all will you do so network runs smoothly?
   (*b*) What all will you do to make sure that data is secure?
   (*c*) What will be your Network Operating Systems for this particular configuration?
   (*d*) In a network arrangement, when you have several different machines, such as system in question, you need to watch for certain factors to keep network trouble-free. What are these factors?
   (*e*) Resource sharing architecture is not suitable for transaction processing in a Client/Server environment. Discuss.
2. What do you understand by Network Management and Remote System Management? How can security be provided to Network?
3. Explain System Administrator in Client/Server Application in detail.
4. Explain in detail, with help of suitable examples, the training advantages of GUI application.
5. Compare and contrast the system administrator training, database administrator training and end user training.
6. What are the responsibilities of the DBA? Also, discuss the capabilities that should be provided by a DBMS.
7. What are different LAN and Network Management issues? Explain with example.
8. Explain the term remote monitoring. What are different network management tools?

8

Client/Server Technology and Web Services

8.1 INTRODUCTION

The Internet and expanded network connectivity established Client/Server models as the preferred form of distributed computing. When talking about Client/Server models of network communication using web services the broadest components of this paradigm become the web browser (functioning as the client) and web server. So, by introducing web services into the equation, Client/Server models become browser/server models. These models are Server-Centric, which make applications easy to load and install, but reduces rich user interaction. Server-Centric applications are currently available from standard browsers, making them convenient and popular with developers. Therefore, a way of enriching user experience is an essential frontier that must be developed for using browser/server models of distributed computing. One of the revolutions of the personal computer was usability or the ease with which humans could communicate with and configure their computers. This usually occurred through individual configuration and the User Interface (UI). Administratively, this was a nightmare because administrators had to install and maintain applications one machine at a time and manage multiple platforms. Individual installation and maintenance across platforms made web services seem like a good solution. Using HTML tools, developers moved toward giving applications global potential and a uniform protocol for management and deployment. The evolving trend was for developers to create applications that run on the server side, while web browsers became, for all intents and purposes, the standard client interface. Client processing power atrophied as execution of programs took place on central servers and output or responses were transmitted back to the browser through standard IP (Internet Protocols). This improved installation, administration, and maintenance. However, to be intelligible to the wide-array of platforms being targeted, web developers had to write in the lowest common denominator or the most widely accepted standards. This affected user experience negatively, while ensuring that applications could be deployed to the most users.

8.2 WHAT ARE WEB SERVICES?

8.2.1 Web Services History

The World Wide Web was developed by Tim Berners-Lee for his employer CERN or the European Organization for Nuclear Research between 1989-1991. In 1990, he wrote the program for the World Wide Web. This program created the first web browser and HTML editor. It was the first program to use both FTP and HTTP.

FTP (File Transfer Protocol) is used to transfer data over a network. Protocol is the set of standards that defines and controls connection, communication, and the transfer of data between two computer endpoints. It determines the format and defines the terms of transmission. HTTP is the protocol that supports hyper-text documents. Both of these protocols are necessary for communication over the Internet or World Wide Web. The source code for the World Wide Web was made public in 1993, making it available to everyone with a computer. The technology continued to develop and between 1991-1994, extended from communication only between scientific organizations, to universities and, finally, to industry. By 1994, computers could transfer data between each other through a cable linking ports across various operating systems.

The first web server, also written by Berners-Lee, ran on NeXTSTEP, the operating system for NeXT computers. The other technology authored by Berners-Lee that is required for Web communication is URLs (Universal Resource Locators). These are the uniform global identifiers for documents on the Web allowing for easily locating them on the Web. Berners-Lee is also responsible for writing the initial specifications for HTML. The first web server was installed in the United States on December 12, 1991 and at SLAC (Stanford Linear Accelerator Center), which is a U.S. Department of Energy laboratory. In 1994, Berners-Lee created the World Wide Web Consortium (WWC) to regulate and standardize the various technologies required for Web construction and communication. It was created to ensure compatibility between vendors or industry members by having them agree on certain core standards. This ensures the ability for web pages to be intelligible between different operating systems and software packages. After 2000, the web exploded. Till date, there exist more than 110 million web sites on the World Wide Web.

8.2.2 Web Server Technology

At the most basic level, the process for web communication works as follows: a computer runs a web browser that allows it to request, communicate and display HTML documents (web pages). Web browsers are the software applications that allow users to access and view these web pages and they run on individual computers. The most popular web browsers are Internet Explorer, Mozilla, Firefox, and Safari (for Mac). After typing in the URL (or address) and pressing return, the request is sent to a server machine that runs the web server. The web server is the program that delivers the files that make up web pages. Every web site or computer that creates a web site requires a web server. The most popular

web server program is Apache. The server machine then returns the requested web page. See the Fig. 8.1(*a*) and 8.1(*b*), depicting the Web Technology yesterday and today.

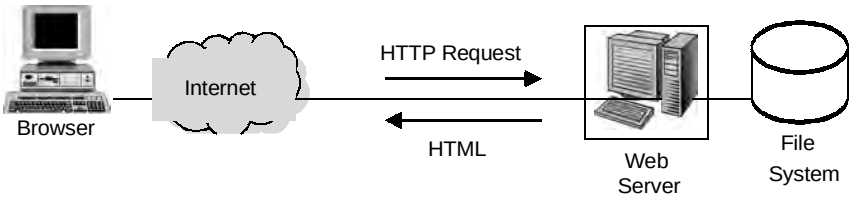


Fig. 8.1(*a*): Web Technology Yesterday

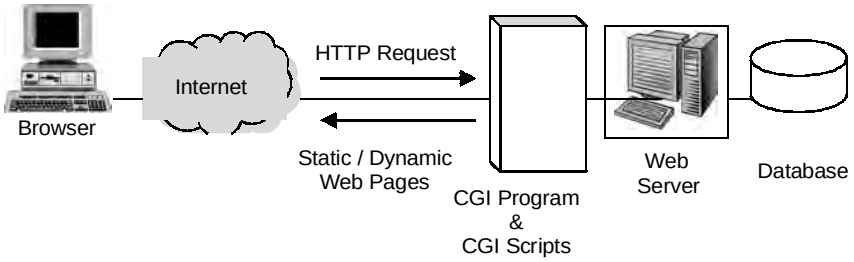


Fig. 8.1(*b*): Web Technology Today

Communication over the Internet can be broken down into two interested parties: clients and servers. The machines providing services are servers. Clients are the machines used to connect to those services. For example, the personal computer requesting web pages according to search parameters (defined by key words) does not provide any services to other computers. This is the client. If the client requests a search from, for example, the search engine Yahoo!. Yahoo! is the server, providing the hardware machinery to service the request. As previously mentioned, each computer requesting information over the Internet requires a web server program like Apache to render the search result intelligible in HTML.

Web servers translate URL path components in local file systems. The URL path is dependent on the server's root directory. The root directory is the top directory of a file system that usually exists hierarchically as an inverted tree. URL paths are similar to UNIX-like operating systems.

The typical client request reads, for example, "http://www.example.com/path/file.html". This client web browser translates this request through an HTTP request and by connecting to "www.example.com", in this case. The web server will then add the requested path to its root directory path. The result is located in the server's local file system or hierarchy of directories. The server reads the file and responds to the browser's request. The response contains the requested documents, in this case, web sites and the constituent pages.

Web Browser (Web Client)

A browser is a software (the most popular web browsers are Internet Explorer, Mozilla Firefox, Safari, Opera, and Netscape) that acts as an interface between the user and the inner workings of the internet, specifically the Word Wide Web Browsers are also referred to as web clients, or Universal Clients, because in the Client/Server model, the browser functions as the client program. The browser acts on behalf of the user. The browser:

- Contacts a web server and sends a request for information.
- Receives the information and then displays it on the user's computer.

A browser can be text-based or graphical and can make the internet easier to use and more intuitive. A graphical browser allows the user to view images on their computer, "point-and-click" with a mouse to select hypertext links, and uses drop-down menus and toolbar buttons to navigate and access recourses on Internet. The WWW incorporates hypertext, photographs, sound, video, etc. that can be fully experienced through a graphical browser. Browser often includes "helper application" which are actually software programs that are needed to display images, hear sounds or run animation sequences. The browser automatically invokes these helper applications when a user selects a link to a resource that requires them.

Accessing Database on the Web Page

Generally, it has been observed that a remote user's web browser cannot get connected directly with database system. But in most of the cases, the browsers are a program running on the web server that is an intermediary to the database. This program can be a common Gateway Interface (CGI) script, a Java servlet, or some code that lives inside an Active Server Page (ASP) or Java Server Page (JSP) document. The program retrieves the information from the page is an ordinary HTML document or the output of some script that Web-based database system. All these activities happen in number of steps explained below and also shown in figure 8.2:

Step1: The user types in a URL or fills out a form or submits a search on a Web page and clicks the Submit button.

Step 2: The browser sends the user's query from the browser to the Web server, which passes it on to a CGI script.

Step 3: The CGI script loads a library that lets it talk to an SQL database server, and it uses that library to send SQL commands to the database server.

Step 4: The database server executes the SQL commands and sends the request information to the CGI script.

Step 5: The CGI script generates an HTML document and writes the HTML document to the Web server.

Step 6: The Web server sends the HTML page back to the remote user.

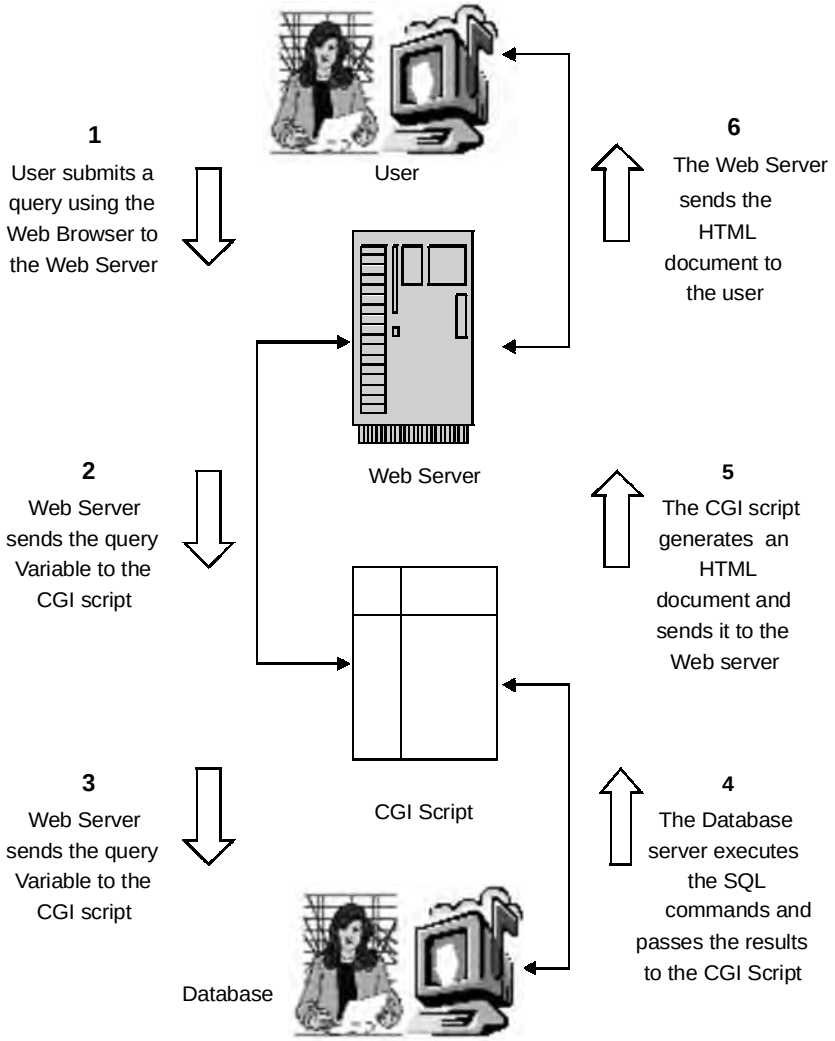


Fig. 8.2: Information Exchange between User and a Web-based Database

If the user has send some information to update a database. Then the CGI Script will generate the appropriate SQL commands and send it to the database server. The database server will execute the SQL commands and then inform the user about the result of execution. A typical example of a database query is search that you perform using a search engine. An example for an insert operation will be filled up a form on your browser to do an online registration for seminar. An example for an update operation will be updating your profile on a portal like naukari.com. In many web sites, even when you type a URL, web pages are generated for you using the information retrieved from a database.

For example, when you browse an online bookshop for a book. There are many details that are dynamic details like price, sales rank, shipping charges, availability, etc. So, it is easy to keep the details about all the books in the shop in a database and generate the web pages as and when requested by the user. The advantage of this is that the changes can be applied to the database and the users always get the up-to-date information.

8.2.3 Web Server

A computer that runs a computer program that is responsible for accepting HTTP requests from clients, which are known as web browsers, and serving them HTTP responses along with optional data contents, which usually are web pages such as HTML documents and linked objects (images, etc.). Although web server programs differ in detail, they all share some basic common features like HTTP and Logging, discussed below.

HTTP: Every web server program operates by accepting HTTP requests from the client, and providing an HTTP response to the client. The HTTP response usually consists of an HTML document, but can also be a raw file, an image, or some other type of document (defined by MIME-types); if some error is found in client request or while trying to serve the request, a web server has to send an error response which may include some custom HTML or text messages to better explain the problem to end users.

Logging: Usually web servers have also the capability of logging some detailed information, about client requests and server responses, to log files; this allows the webmaster to collect statistics by running log analyzers on log files.

In practice many web servers implement the following features also:

- Authentication, optional authorization request (request of user name and password) before allowing access to some or all kind of resources.
- Handling of not only static content (file content recorded in server's filesystem(s)) but of dynamic content too by supporting one or more related interfaces (SSI, CGI, SCGI, FastCGI, JSP, PHP, ASP, ASP.NET, Server API such as NSAPI, ISAPI, etc.).
- HTTPS support (by SSL or TLS) to allow secure (encrypted) connections to the server on the standard port 443 instead of usual port 80.
- Content compression (i.e., by gzip encoding) to reduce the size of the responses (to lower bandwidth usage, etc.).
- Virtual hosting to serve many web sites using one IP address.
- Large file support to be able to serve files whose size is greater than 2 GB on 32 bit OS.
- Bandwidth throttling to limit the speed of responses in order to not saturate the network and to be able to serve more clients.

Although web servers differ in specifics there are certain basic characteristics shared by all web servers. These basic characteristics include HTTP and logging. As previously, mentioned HTTP is the standard communications protocol for processing requests between

client browsers and web servers. This protocol provides the standardized rules for representing data, authenticating requests, and detecting errors.

The purpose of protocols is to make data transfer and services user-friendly. In computing, the protocols determine the nature of the connection between two communicating endpoints (wired or wireless) and verify the existence of the other endpoints being communicated with. It also negotiates the various characteristics of the connection. It determines how to begin, end, and format a request. It also signals any errors or corruptions in files and alerts the user as to the appropriate steps to take. HTTP is the request/response protocol used specifically for communicating. HTML documents which is the language hypertext or web pages are written in. However, responses can also return in the form of raw text, images or other types of documents. The other basic web server characteristic is logging. This is a feature that allows the program to automatically record events. This record can then be used as an audit trail to diagnose problems. Web servers log detailed information recording client requests and server responses. This information is stored in log files and can be analyzed to better understand user behavior, such as key word preferences, generate statistics, and run a more efficient web site.

There are many other practical features common to a variety of web sites. Configuration files or external user interfaces help determine how much and to what level of sophistication users can interact with the server. This establishes the configurability of the server. Some servers also provide authentication features that require users to register with the server through a username and password before being allowed access to resources or the execution of requests. Web servers must also be able to manage static and dynamic content. Static content exists as a file in a file system. Dynamic content is content (text, images, form fields) on a web page that changes according to specific contexts or conditions. Dynamic content is produced by some other program or script (a user-friendly programming language that connects existing components to execute a specific task) or API (Application Programming Interface the web server calls upon). It is much slower to load than static content since it often has to be pulled from a remote database. It provides a greater degree of user interactivity and tailor responses to user requests. To handle dynamic content, web servers must support at least any one of interface like JSP (Java Server Pages), PHP (a programming language that creates dynamic web pages), ASP (Active Server Pages) or ASP.NET (it is the successor of ASP).

8.2.4 Web Server Communication

Web servers are one of the end points in communication through the World Wide Web. According to its inventor, Tim Berner-Lee, the World Wide Web is "*the universe of network-accessible information, an embodiment of human knowledge.*" The World Wide Web is the global structure of electronically connected information. It refers to the global connections between computers that allow users to search for documents or web pages by requesting results from a web server. These documents are hyper-text based (written in HTML-Hypertext Markup Language), allowing users to travel to other pages and extend their

research through links. They are delivered in a standardized protocol, HTTP (Hypertext Transfer Protocol, usually written in lower case letters), making HTML documents intelligible across hardware and software variations.

This information travels through web servers and web browsers. The communication initiates from a user request through a web browser. The request is delivered to a web server in 'HTTP' format. The server then processes the request, which can be anything from a general search to a specific task, and returns the results in the same format. The results are written in HTML, which is the language web pages are written in that supports high-speed travel between web pages. HTML is also essential for displaying many of the interactive features on web pages, such as linking web pages to other objects, like images. An important distinction when defining web servers is between hardware and software. A web server is also a computer program (software) that performs the functions outlined above.

8.3 ROLE OF JAVA FOR CLIENT/SERVER ON WEB

Client server models provide the essential mechanisms for working with the Internet. In fact, most of the World Wide Web is built according to this paradigm. In client server models the web browsers run by millions of users are the clients. On the other side of the equation, is the web hosting systems that run at host sites and provide access to processes and data requested by the client. In this case, these hosting systems are the server. This definition is based on software programs, where the client is a program running on a remote machine that communicates with a server, a program running at a single site and providing responses to client requests, such as web pages or data.

Java is a programming language that has been developed specifically for the distributed environment of the Internet. It resembles C++ language, but is easier to use. C++ is a high level programming language that performs low level functions. In computing, low level functions are those that focus on individual components and the interaction between them as opposed to abstracted and systemic features (high level). Java, like C++, is a language that is multi-paradigm and supports object-oriented programming (OOP), procedural programming, and data abstraction. Object-oriented programming is increasingly being used in client server technology. It refers to a programming language model that is organized by objects rather than actions, data rather than logic. OOP identifies objects (sets of data) and defines their relationship to each other. These objects are then generalized into a class. Methods or sequences of logic are applied to classes of objects. Methods provide computational instructions and class object features provide the data that is acted upon. Users communicate with objects and objects with each other through specifically defined interfaces called messages. Java can create applications that are distributed between clients and servers in a network. Java source code (signaled by .java extension) is compiled (source code transformed into object code) into byte code (signaled by .class extension). A Java interpreter then executes this byte code format. Java interpreters and runtime environment run on Java Virtual Machines (JVMs).

The portability and usability that characterizes Java stems from the fact that JVMs exist for most operating systems, from UNIX, to Windows, to Mac OS.

Java is one of the most well-suited languages for working on the World Wide Web and the client server model is the primary models for working on distributed networks, of which the World Wide Web is just one. There is natural affinity between the two and this article will discuss some major characteristics of Java and how it can be utilized in building client server systems.

To understand how Java is used in client server systems it becomes essential to understand the major characteristics of Java. Syntactic of Java are similarity to C and C++ languages, Java is simple, simpler, in fact, than the languages it emulates. Java is also a robust programming language, which means it creates software that will identify and correct errors and handle abnormal conditions intelligently. Another major characteristic of Java is that it is object oriented programming, which was described above. OOP is characterized by three properties also present in Java programming: inheritance, encapsulation, and polymorphism. Inheritance is a major component in OOP which defines a general class and then specializes these classes by adding additional details in the already written class. Programmers only have to write the new features since the specialized class inherits all the features of the generalized class.

In OOP, encapsulation is the inclusion of all methods and data required for an object to function. Objects publish their interfaces and other objects communicate with these object interfaces to use them without having to understand how the encapsulated object performs its functions. It is the separation of interface and implementation. Polymorphism in OOP in general and Java specifically, is the ability to assign a different set of behaviors to an object in a subclass from the methods describe in the more general class. Therefore, subclasses can behave differently from the parent class without the parent class having to understand why for change itself. Multi threading is also an important characteristic of Java that increases interactive responsiveness and real time performance. Threading is the way a program splits or forks itself into two or more tasks running simultaneously. This allows for thread based multi tasking. Multi threading creates the effect of multiple threads running in parallel on different machines simultaneously.

Socket-based Client Server Systems in Java

Java builds client server systems with sockets. Sockets are the endpoints of two-way communication between programs running in a network. They are software objects that connect applications to network protocols, so they become intelligible. For example, in UNIX a program opens a socket that enables it to send and receive messages from the socket. This simplifies software development because programmers only have to change or specify the socket, while the operating system is left intact. In client/server models, the server contains sockets bound to specific port numbers. Port numbers identify specific processes that are to be forwarded over a network, like the Internet, in the form of messages to the server. The server only has to monitor the socket and respond when a client requests

a connection. Once connections have been made and bound to port numbers, the two endpoints, client and server, can communicate through the socket. Java sockets are client side sockets, known simply as sockets and server side sockets known as server sockets. Each belong to their own class within a Java package. The client initiates the socket, which contains a host name and port number, by requesting connections.

The server socket class in Java allows for servers to monitor requests for connections. As the socket communicates over the network, Java's server socket class assigns a one port number to each client in the server and creates a socket object or instance. This server socket instance creates a connection and produces a thread or string of processes through which the server can communicate with the client through the socket. Java web servers are built according to this model. TCP (Transmission Control Protocol) works in conjunction with IP (Internet Protocol) to send messages between computers over the Internet. When communicating over the Internet, the client program and the server program each attach a socket to their end of the connection. Then the client and server can read from and write to the socket. Java provides operating system sockets that allow two or more processes to send and receive data, regardless of computer location.

Java's RMI System

The other method for using Java to build client server systems is RMI. RMI stands for Remote Method Invocation. By using Java language and functions, programmers write object oriented programming, so that objects that are distributed over a network can interact with each other. RMI allows for client objects in JVMs to request services through a network from another computer (host or server). Client objects included with the request may call upon methods in the remote computer that change the results. Methods are programmed procedures attributed to a class that are contained in each of its objects (or instances). It is a characteristic of object-oriented programming.

Classes and, therefore, objects can have more than one method and methods can be used for more than one object. Responses then run as if they were local (on the same computer.) This passing back and forth of objects and methods attached to objects is called 'object serializations'. Simply put, RMI requests call upon the method of a remote object. As previously stated, it uses the same syntax it would locally. To make this intelligible to the servers or sites being called upon requires three layers: a client side stub program, a remote reference layer, and a transport connection layer. Each request travels down the layers of the client computer and up the layers of the server. The client side stub program initiates the request. Stub programs are small sections of programs containing routines from larger programs. It is a substitute for programs that may take too long to load or are located remotely on a network. They are provided by the server at the client's request. Stubs accept client requests and communicate requested procedures (through another program) to remote applications. They also return the results to the client or requesting program. These stubs mimic the program being called for service.

In Java, stub programs are also referred to as 'proxy'. Remote reference layers manage reference variables for remote objects, using the transport layer's TCP (Transmission

Control Protocol) connection to the server. Reference variables contain class data and therefore include methods. The transport layer protects and maintains end to end communication through a network. The most used transport layer is TCP. After the client requests pass through the transport layer, they pass through another remote reference layer before requesting implementation of the request by the server from a skeleton. These skeletons are written in high level IDL (Interface Definition Language). The server receives the request and sends the response along the same channels, but in the other direction.

8.4 WEB SERVICES AND CLIENT/SEVER/BROWSER—SERVER TECHNOLOGY

Web services and Client/Server technology made it possible for applications to integrate of separate components. These components might exist on separate machines, but they work together through network (Internet) communication. Applications using web services demonstrate the integration of components coming from multiple sources. This makes version management important. One of the benefits of having to focus on version management is to make developers aware of component dependencies and specific areas that require maintenance in each version. This allows developers to customize maintenance for application deployment. These distributed application components use universal formats provided by such programming languages as XML (Extensible Markup Language) and WSDL (Web Standard Description Language).

XML is the W3C standardized language that allows information and services to be written in a structurally and semantically intelligible way that both humans and machine on different platforms can understand. It can be customized with user or industry tags. WSDL uses an XML format to describe network services. These services are described as endpoints that use messages to transmit documents or procedure-oriented information. These operations and messages are abstract, but then they are attached to specific network protocols and message formats to enable communication. Distributed application components also use universal protocols like HTTP (Hypertext Transfer Protocol) and SOAP (Simple Object Access Protocol).

HTTP is the standard protocol for transmitting HTML files or documents. SOAP is a message-based protocol formatted in XML and using HTTP. It is a way for a program running in one operating system to communicate with a program running in another. For the purposes of this discussion, applications that take advantage of web services will be understood as ‘balanced computing model’ because these applications are designed to take the fullest advantage of both client and server capabilities in the most efficient way. This model of balanced computing improves traditional Client/Server model by not stressing one part of the system and ignoring the capabilities of other parts.

For example, traditional browser-server models were Server-Centric. They could handle user demands but did not take advantage of client-side processing that could predict user behaviour. To predict the behaviour from a diverse customer base requires headroom. Headroom is also known as the attenuation crosstalk ratio. It ensures the network

connections are strong and that signals on the receiving end of a communication are strong enough to overcome any interference. This provides a consistent and customized user experience regardless of unpredictable behaviour in the network.

8.5 CLIENT/SERVER TECHNOLOGY AND WEB APPLICATIONS

There is a gap in user experience between desktop applications and web applications. Desktop applications run on a single computer, while web applications run on the Internet. Since the invention of the Web, developers have been trying to design web applications that demonstrate the speed and interactivity of applications running on the client machine of a LAN (Local Area Network). Despite the explosion of web based applications in the 1990's (and continuing today), many users still prefer desktop applications. Like web sites, desktop applications access up to date information by connecting to the Web through the Internet.

However, desktop applications are designed with a much more refined sensibility and use PC power to customize requests from information stored on the desktop. This user experience is significantly better than when using remote web sites. Many are arguing that desktop applications will be the next wave in the Internet revolution.

So, with desktop applications breathing down their necks, web applications need to keep up the pace. Web applications are accessed by web browsers and run over a network, like the Internet. They are popular because of the power dominance of web browsers serving as thin clients.Thin clients are client applications or devices that simply initiate requests and provide input. They do very little of the processing, letting the server handle the heavy lifting by forwarding requests and contacting different nodes and networks. Web applications are responsible for web based e-mail applications like Hotmail, online shopping sites, online auction sites, wikis, and blogs. In traditional client server computing, every application contained a client program that provided the User Interface (UI) through which users would interact with/make requests from the applications. Each client program had to be installed individually on each user workstation. Web applications are different.

Web applications dynamically generate web documents. These are HTML/XHTML (Hypertext Markup Language/Extensible Hypertext Markup Language) documents transmitted over the Internet through HTTP (Hypertext Transfer Protocol). These documents or pages make up the web site. Using a standard server side scripting language like JavaScript allows for more interactive features in the user interface. Usually, each page is delivered as a static document, but the sequence in which they are presented is interactive. User web forms are embedded in the page markup, customizing responses. The web browser, acting as "universal client", interprets and displays the dynamic web pages. Web application UIs offer a variety of features. For example, the drag and drop feature allows for virtual objects to be moved from location through the mouse. This can create all sorts of actions, from copying to associating two objects to create a new action. By using application specific technologies like Java, JavaScript, DHTML (Dynamic HTML), and Flash, all sorts of graphic

and audio interactive features may be added to a UI. As previously stated, web developers are currently looking for ways to improve user experiences with web applications so they may closely resemble the performance of desktop applications. Remember, the user experience is the most often gauged by the usability of the UI or GUI (Graphic User Interface).

1st Generation Web Applications

The applications that are available now are typified by the technology used/presented in the 1st Generation Web Application; see the Fig. 8.3 shown. This might be characterized as a new form of electronic publishing, but it's richer in some ways than books because it's multimedia publishing. Today most home pages consist of text, photos, and graphics. By early 1997, however, it's likely that animation and 3D applications will be available. This technology is already very useful in information dissemination. Companies are replacing human resources manuals and maintenance manuals with browsers connected over intranets or the Internet to a server which contains the latest information sought.

A primary limitation of first generation applications is that there is no database management system, connected to the web server and the software does not keep the track of who is requesting information or of the last request from that user. It is a stateless connection. The addition of DBMS capabilities to the HTML processes on the server will allow HTML servers to have memory. As the leading DBMS vendors add connections for Web servers, it becomes possible for that server to remember who you are and what you have done from page to page and from visit to visit. The interaction, then, becomes a lot more intelligent and useful.

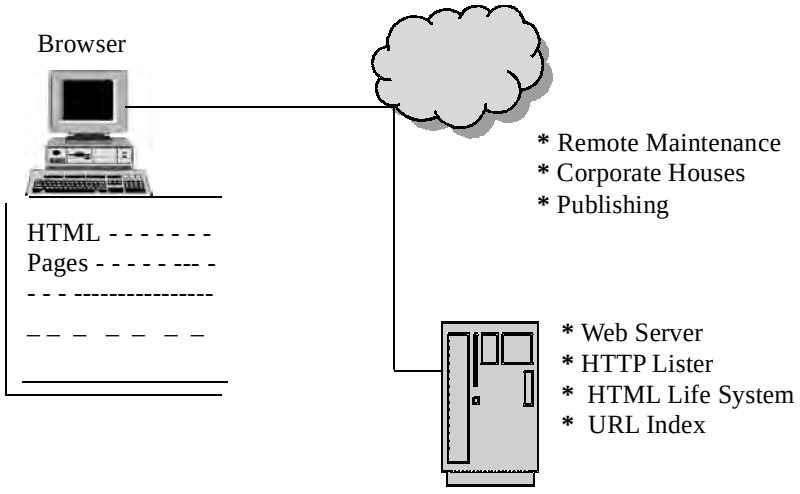


Fig. 8.3: Architecture of Ist Generation Web Application

2nd Generation Web Applications

First Generation Web Applications are quickly going to be joined by newer more capable environments that perhaps we can call the second generation. Several things will define this newer generation that are given as below:

- Support for active clients via downloadable applets (software components),
- Live DBMS links that enable the server to know who you are from page to page, and visit to visit, and
- True interactivity between the client and server (indicated by the two-headed arrow).

2nd generation Web applications have live DBMS connections on the server. Today what is mostly available for such support are SQL DBMS engines from companies like Sybase, Informix, IBM and Oracle. A problem with these engines is that SQL supports traditional business data types such as alphanumeric, date and time. No major SQL product supports extended and complex data types such as voice, maps or video at this time. That is a defect that will be remedied (probably) in the 1997 timeframe. All of the major DBMS vendors are making important strides in this direction.

This software component type of operation will be the first example of widespread use for distributed object computing. Sun's Java technology is the best known example of this approach. Sun describes its Hot-Java browser/Java language technology as "simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high performance, multi-threaded and dynamic." The way this will work is for your browser on the client to have a Java (or ActiveX or C++) interpreter that can activate a component that has been downloaded to your client from the Web server. Your browser, then, becomes event driven and can exhibit various types of behavior.

From a software point of view, we will see both "inside/out" and "outside/in" approaches to write the code to mix applets and normal programming environments. BASIC compilers, for example, will be extended to support embedded HTML code. And, HTML will be extended to handle embedded Java, ActiveX and other component technologies.

In the Java environment pointers are not supported and that makes it impossible for any downloaded applet to address segments of memory outside the virtual Java machine that has been created inside your browser. This enforces security for your client and makes sure that any downloaded applets don't behave in a malicious fashion.

While talking heads and multimedia demos have been used to illustrate the operation of the Java/Hot Java environment, the real benefit to corporate users will come from a new type of application that hasn't been possible before–collaborative computing among strangers. Your server based applications are now available to everyone in the world; your programs can execute on their clients. If you want to do group scheduling, for example, until now everyone in the effort had to have the same client, something like Lotus Notes. With this new environment, it will be easy to accomplish wide area, multi-platform group scheduling via the Internet. The scheduling applet can be downloaded and executed in the component enabled browser.

8.6 BALANCED COMPUTING AND THE SERVER'S CHANGING ROLE

In balanced computing, processing is shared between clients, servers, and any other devices connected via a network. This distribution is inherent in the design since applications are already spread out on different machines and connected through web services. In balancing, the processing required by each new use is often shifted back to the user's system, thereby taking fuller advantage of client-side processing power. This allows for improved scalability, since the processing load is increased insignificantly by the addition of users.

Load balancing can also be achieved by building Service-Oriented Applications (SOAs) where components run on different nodes in multiple locations duplicate services on multiple nodes. This duplicating of services on multiple nodes also improves reliability since there is no single point of failure that will topple the entire application. Through balanced computing, platforms can take maximum advantage of computing and display capabilities on each platform. The virtual application which is balanced across multiple nodes remains transparent (its complex functions hidden) while the user utilizes his or her own collection of devices to run and view the application on the user end.

Balanced computing not only distributes the processing load, but changes the role of the server as well. Instead of computing so heavily, the server primarily directs traffic. Given rich clients and decent Internet connectivity, users directly contact databases instead of requiring server intervention. Rich clients are applicants in user computers that retrieve data through the Internet. As previously discussed, the proliferation of web-based applications replaced the user interface with the web browser. Scripting languages, like JavaScript or VBScript, were embedded into HTML to improve user interfaces (among other things). Java applets were also added. But nothing could compete with the user experience of using an application built from its local environment. Developing technologies like improved web-page scripting languages and AJAX (Asynchronous JavaScript and XML), made web browsers function more like rich client applications.

Another method used to reduce demands on the server uses a connecting device to re-direct previously server-side processing to the client. This depends on the client device capability and Internet connectivity. If these are weak, the server picks up the slack, making application performance consistent on both the client and server ends.

User Experience and Development

Balanced distributed computing models improve user experience and expand development. Developers can focus on user experience by examining devices and customizing features. For example, different user interfaces can be customized for different departments that require different resources to perform their function. Different roles would have their own user interface. Well-defined user roles and profiles that are stored on user machines make more efficient use of server computation. It allows the server to pull customized responses based on the identity/role of the user. To further reduce server demand, clients can communicate directly with databases by requesting information for

the user role profile. This eliminates the web server as middleman for the request and computation on the output side.

Data integration on user platforms offers new opportunities to build applications that draw data from a variety of sources and can add different contexts. In a balanced distributed computing model, web services send information that is usually stored on databases or servers (like financial information) to the user's machine. It accomplishes this by using the client-side's processing power. These responses are formatted in the increasingly popular, universal XML. Desktop applications (on the user's system) can take that information and analyze it in different contexts.

The decentralization of distributed browser-server models also improves security and protects privacy. For example, data repositories are often located in a different location from the server. This makes it more difficult for external attackers to find. It also makes it less accessible to internal attackers. Also, it is safer for user profiles to be stored on individual machines, rather than on a central database.

Distributed computing models address the future of IT architecture and application. Organization must aim to create independent and flexible applications that can respond quickly to a variety of contexts. Connections must be agile. Loosely coupled applications, characteristic of distributed computing models, withstand broken connections and slow Internet performance. This protects core technologies from customer demands and lack of Internet bandwidth.

EXERCISE 8

1. Explain an object web model based on java client and CORBA ORB's on the basis of following points:
   (*i*) Web client
   (*ii*) Protocol used
   (*iii*) Web server
2. Explain end-to-end working of Client/Server web model. (Hint: Use CGI, GET, POST, Web browser and Web server)
3. Explain, with the help of block diagram and example, the Common Object Request Broker Architecture.
4. Discuss the role of traditional and web databases in handling Client/Server based applications.
5. Discuss the role of web browser for providing web service in Client/Server environment.
6. Explain how the Database is being accessed on the Web. Or how the information exchange take place between User and a Web-based Database.
7. Discuss the development of Client/Server Technology based web applications.

9

Future of the Client/Server Computing

9.1 INTRODUCTION

Development of Client/Server technologies is still evolving. From year to year, project-to-project, technocrats are not doing anything the same way twice. Today, we are busy moving our Client/Server assets to the intranet. In a few years, who knows where we'll take Client/Server and distributed development? Spotting trends is not only a nice thing to do, it's a career skill. For instance, those who saw the rise of the intranet early on, and obtained the skills they needed to leverage the technology, were the same people who cashed in when the Web took off. The same can be said about three-tier Client/Server, and now about the rise of distributed objects. We may also see the rise of user agents, or point-to-point networking.

In this chapter, we shall discuss the bright future of Client/Server technologies, including technology that will be seen in the near and distant future. Furthermore, we will discuss about some of the trends that you can latch on today, and where to catch the next wave of Client/Server technology.

9.2 TECHNOLOGY OF THE NEXT GENERATION

Predicting technology is like predicting the weather. While it's safe to say that processors will be faster and disk space cheaper, it's much more difficult to predict the way we'll use and develop software. Developers don't create the trends, they follow them. For example, the interest in the intranet came from the millions of users who found a friendly home in their Web browser, and wanted a way to run their business applications with the Disney and CNN Home Pages. The same could be said about traditional Client/Server computing a few years ago. Users wanted to run business applications with their new GUI desktop applications.

Using the past as our guide, we can make an intelligent gives about the future of Client/Server computing. The predictions can be broken up into a few categories: networking, development tools, processors and servers, paradigms, and enabling technologies.

9.2.1 Networking

The pipe between the client and server is still too narrow, and bandwidth has not kept up with the development of technology and modern architecture. With the advent of ATM and switched networks, we can finally count on a pipe wide enough to push tons of data through. It will take a few years before we bring this wide pipe down to the desktop.

Client/Server developers must become networking gurus as well. The performance of the network dictates the performance of the Client/Server system. What's more, with the advent of application-partitioning technology (such as application-partitioning tools and distributed objects), the network not only links the client to the server, but links all the application objects together to form the virtual (partitioned) application. Clearly, the network is the infrastructure of the distributed application.

In addition to upgrading the speed and reliability of enterprise network technology, we are looking for ways to upgrade the speed of WAN technology. Frame relay and other global networking solutions will create high-performance virtual systems, available throughout the world. Let us hope this technology will extend to the Internet. If you haven't noticed, it's creaking under the strain of thousands of additional users, who start surfing every day.

9.2.2 Development Tools

Client/Server development tools are finally delivering on promises made initially, but there is a lot of room for improvement. Some of the areas where tools can be made to perform better include:

- Use of true compilers.
- Native links to the distributed objects and TP monitors.
- Better component development capabilities.
- Use of standards.
- Consistent language support.
- True application-partitioning capabilities.
- Consistent approach to the intranet.

Use of true compilers: With the advent of Delphi, developers saw the benefit of a specialized development tool that cannot only do RAD, but create small, efficient, and speedy applications. The use of a true compiler allows developers to create native executables and avoid the overhead of dealing with an interpreter.

In the past, specialized Client/Server development tools counted on the fact that processors increased in speed every year to mask the inefficiencies of their deployment mechanisms. But users did notice, and they labeled PowerBuilder, Visual Basic, and other

specialized development tool applications "dogs" on low-end PCs. Upgrading the hardware in the entire company to run a single application costs millions, and there is something to be said about efficient application development (such as is offered by most C++ development environments).

Today we see a trend in specialized Client/Server development tools that offers a true compiler. Besides Delphi, PowerBuilder runs close to native, and Visual Basic (version 5) will also provide a true compiler. Other specialized tool vendors are bound to head in that direction. Tool vendors should have done this from the start, and it's about time they got their act together.

Native links to the distributed objects and transaction processing monitors: Despite the fact that distributed objects and TP monitors are key enablers for multi-tier Client/Server development, few Client/Server tools exist that can easily use them. For instance, if we want to use a specialized Client/Server tool with a TP monitor, we have to load and invoke the services of a DLL (in most cases), or limit the selection of tools to those few that support TP monitors (e.g., Prolific, EncinaBuilder). The same can be said about distributed objects.

As mentioned above, the number of multi-tiered Client/Server applications keep growing, and so will the need for Client/Server tools that can access the services of TP monitors and distributed objects. With demand comes innovation, and most tool vendors plan to provide links to the distributed objects and TP monitors. With the advent of Transaction Server, for example, TP monitors come as close to a DCOM connection as any COM-enabled Client/Server development tools.

As IIOP makes intranet development easier, we'll see a rise in the number of tools that support traditional Client/Server development and integration with distributed objects. Thus, as a byproduct of links to Web technology, we'll see the use of CORBA-compliant distributed objects as a part of most Client/Server development tools. Already, any Windows-based Client/Server development tool that does OLE automation can also link to DCOM, and thus, COM-enabled ORBs. The movement toward the use of distributed objects will continue.

Better component development capabilities: The other side of distributed objects is the ability to assemble Client/Server applications from rebuilt components. While most Client/Server tools support the integration of components (e.g., ActiveX or Java), they don't support them well. Many components don't work and play well with others, and don't provide developers with enough granularity.

If component development is to work, tool vendors must provide consistent support for components that will allow developers to modify the interfaces, and easily link components together to create an application. What's more, the same tools should create components. We have many examples of tools today (such as Visual Basic and PowerBuilder) that can both create and use components. The future lies in tools that can easily create and share components, as well as mix and match tools to construct an application from a very low level (e.g., middleware) to a very high level.

If current indicators continue to hold true, ActiveX will provide the component standards we need for Windows, while OpenDoc will have limited success on non-Windows platforms. A lot will depend on Microsoft's ability to parlay ActiveX into a legitimate open standard. Right now, developers view ActiveX as a proprietary standard, still bound to Windows. They are right.

Use of standards: Of course, the use of distributed objects, TP monitors, and components leads to a discussion of standards. While many standards and standards organizations exist today, standards are only as good as the number of developers who use them.

Key Client/Server standards include CORBA, COM, and SQL92, but many others run with the wolves. A common problem in the industry is our failure to use and enforce standards. The tradeoff is the exclusive advantage that vendors enjoy while they employ proprietary features versus the benefits they could reap if they would provide developers with standards they support in other tools. While many Client/Server technology vendors give lip service to the idea, standards are not yet a priority.

The movement toward standards really depends on the users/developers. If we demand that tool and technology vendors employ standards, and agree that interoperability is of great value, the vendors will respond. The standards organizations (such as OMG) also need to work harder to bring standards to the technology. It took five years before the OMG had a standard that actually spelled out a way for CORBA-based distributed objects to work together. That's just too long for this business.

Despite the slow movement, I think we'll continue to move toward a standard technology that will let everyone and everything work together. The trick now is to pick the standard(s) you think will win.

Consistent language support: Until recently, the mantra of Client/Server tool vendors was 'build a tool, build a proprietary language.' Fact is, we have more development languages today than ever before, with languages proprietary to particular tools. The reasons are the same as with our previous standards discussion.

The most recent trend is for Client/Server tool vendors to employ non-proprietary languages in their tool, or to use the same language throughout a product line. For example, Delphi is based on Pascal, rather than a proprietary scripting language like PowerBuilder. Optima++ and VisualAge C++, use C++ as their native language. Visual Basic shares VBA with Access, Microsoft Office products, and even third-party tools such as Oracle's PowerObjects. VBA is licensed by Microsoft to over forty vendors. This trend will continue. While developers are happy to learn a new IDE, they aren't nearly as thrilled when they must learn a new programming language.

True application-partitioning capabilities: Along with links to the distributed objects, tools will continue to provide more sophisticated proprietary application-partitioning capabilities. Many traditional two-tier tools, such as Power Builder and Unify, are heading in this direction (albeit slowly), while Forte, Dynasty, and IBI's Cactus are learning to do a better job with their existing partitioning tools.

There is also a movement in the application-partitioning world toward the use of open technologies. For instance, distributed objects and TP monitors now work with the proprietary ORBs of application-partitioning tools. Proprietary ORBs are not a long-term solution, and the opening of these environments to non-proprietary technology will only increase their value to developers.

Consistent approach to the intranet: The enabling technology of the intranet must settle down to a few consistent approaches. For example, now we have HTML, SGML, VRML, CGI, NSAPI, ISAPI, Java, JavaScript, VBScript, ActiveX, Java, IIOP, and the list goes on. Although this provides developers with an arsenal of technologies and techniques, it results in a few too many standards to follow, and confusing for developers.

Over the next year, we'll see the list of intranet-enabling technologies shorten, as the market naturally removes the technologies that do not capture the hearts and minds of the developers and that offer redundant technologies. Redundant technologies include Java and ActiveX, JavaScript and VBScript. We'll also see a movement toward ISAPI and NSAPI, or back to CGI. Finally, we need to go with a single HTML standard, and move away from the proprietary extensions of Netscape and Microsoft.

9.2.3 Processors and Servers

We can expect processing power to increase; without any slowdown in that area. The power of servers will be keeping up increase in future, up with the requirements of your application, and we can now run mainframe class systems on commodity hardware.

We'll also see the rise of symmetric multi-processing computers and operating systems for use as clients as well as servers. When Windows 95 and Windows NT merge, clients will have a powerful operating system that can make the most of this new hardware. Clients can once again become a location for application processing.

Servers will become more component-based. Architects will be better able to customize servers around particular application server and database server requirements, adjusting the cache, memory, processors, and disk size to the exact specifications of the application. The operating system that will run these servers will have to keep up. Advanced multi-processing operating systems (such as Windows NT and Unix) will provide better load balancing and fault-tolerant capabilities, including, for instance, the ability to better work through memory, disk, and processor failures without service interruptions.

Despite the religious implications of operating systems and an anti-Microsoft sentiment, Windows NT will take more market share away from the traditional server operating system for Client/Server: Unix. Windows NT is almost as powerful, supports multi-processing, and can run on a number of processors. What really drives the movement toward Windows NT is the ease of use it offers, as well as its ability to support the majority of off-the-shelf software. While Sun servers will run Oracle, they won't run Word for Windows as a native application. Web servers for use on intranets or the Internet will become the domain of NT as well as Microsoft is giving its Web server away with NT, which is a convenient situation.

9.2.4 Paradigms

Today we are well into the object-oriented development paradigm, and this will remain true. In fact, as we become better at using object Client/Server tools, we will dig deeper into their support for the object-oriented development model.

The use of components will become more of an issue too. We really can't build an application by mixing and matching today's components. However, as component standards finally take off, we'll be able to snap in many application components, and even mix and match components with the native objects of the tools.

9.3 ENABLING TECHNOLOGY

We must consider the evolution of the enabling technologies of Client/Server. Enabling technologies are the combinations of hardware and software that can be used to assist in creating a particular kind of application system. These technologies include TP monitors, databases, and middleware, Expert systems, Point-of-Services (POS), imaging, intranet and extranet.

Transaction-processing monitors: As we move headlong into large-scale, mission-critical distributed computing, we need an "ace in the hole." TP monitors are that ace. Now developers have begun to understand the benefits of the TP monitors with other alternatives (such as proprietary application-partitioning tools) proved themselves to be a bit less popular. With the advent of Microsoft's Transaction Server, we now have a TP monitor that fits easily into commodity Windows environments, built from the ground up for ActiveX and the intranet. With the success of Transaction Server, We may see more TP monitors heading into the market.

Databases: Databases will continue to dazzle us with better performance and new features. The big three players (Oracle, Sybase, and Informix) will continue to dominate share, and the relational database model will remain. The concept of the universal server will enlarge the database market by allowing databases to be a "jack of all trades" for developers (object-oriented, relational, Web, binary, etc.). Databases vendors will find, as Oracle is finding now, that working with distributed objects provides a competitive advantage as the rest of the world moves there.

Middleware: Finally, middleware will evolve too. While middleware is powerful and necessary, it's difficult to use and deploy. In the next few years, we'll see a rise of middleware solutions—both message and RPC-based—that provide "snap-in" functionality and consistent interfaces. Microsoft's Falcon message-oriented middleware will once again prove that Microsoft can turn a high-end product into a consumer commodity, and other, more aggressive vendors will have to keep up.

9.3.1 Expert Systems

Expert system is a branch of artificial intelligence that makes extensive use of specialized knowledge to solve problem at the level of human expert. Expert systems are intelligent

computer programs that use knowledge and inference process to solve problems that are difficult enough to require significant human expertise for their solution. That is, an expert system is a computer system that emulates the decision-making ability of a human expert. The term expert system is often applied today to any system that uses expert system technology that includes special expert system languages, programs and hardware design to aid in the development and execution of expert systems. The knowledge in expert systems may be either expertise or knowledge that is generally available from books, magazines, and knowledgeable persons. The terms expert system, knowledge based system, or knowledge based expert system are often used synonymously. Fig. 9.1 given below, illustrates the basic concepts and the architecture of knowledge based expert system that consists of two main components internally. The knowledge base contains the knowledge based on which the *inference engine* draws conclusion.

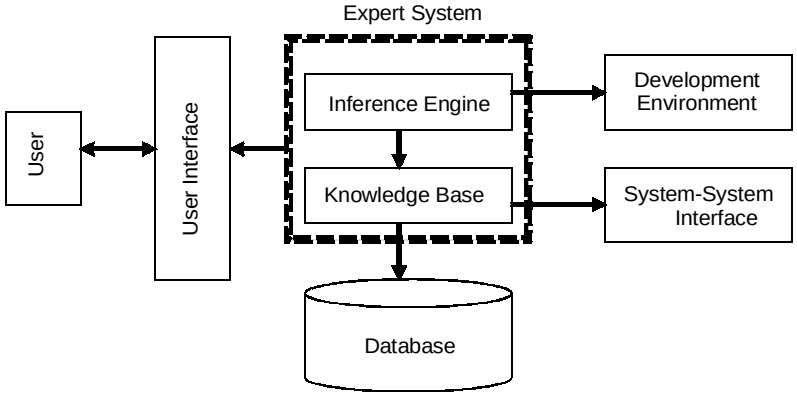


Fig.9.1: Basic Concept of Expert System

Applications of expert system are widely accepted and well-suited for the Client/Server models. The *user interface* provides some rule based advantages related with the processing power and some of the benefits at the workstations. Mostly the rules are managed by knowledgeable user and not by a professional programmer, because the user only knows how his job works (a job the expert system emulates). The inference engine, a CPU-intensive process that takes advantage of low-cost processing and RAM available with Client/Server technology, enforces the rules. Most applications will be implemented using existing databases on host-based DBMS's. The Client/Server model provides the necessary connectivity and services to support access to this information.

Expert system provides an advantage in business world by encapsulating the rules and object of the trade. Objects are created by expert knowledge hence can be reused throughout the organization. The outcomes of the expert system can frequently be used in integrated business systems.

9.3.2 Imaging

Widely used digital documents are imaging, structured documents, and distributed hypertext, active or compound. Fig. 9.2 given below, illustrates the various forms of existing digital documents).

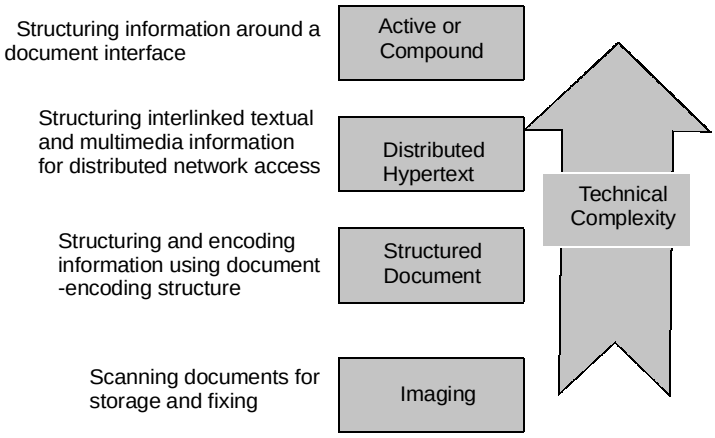


Fig. 9.2: Types of Digital Documents

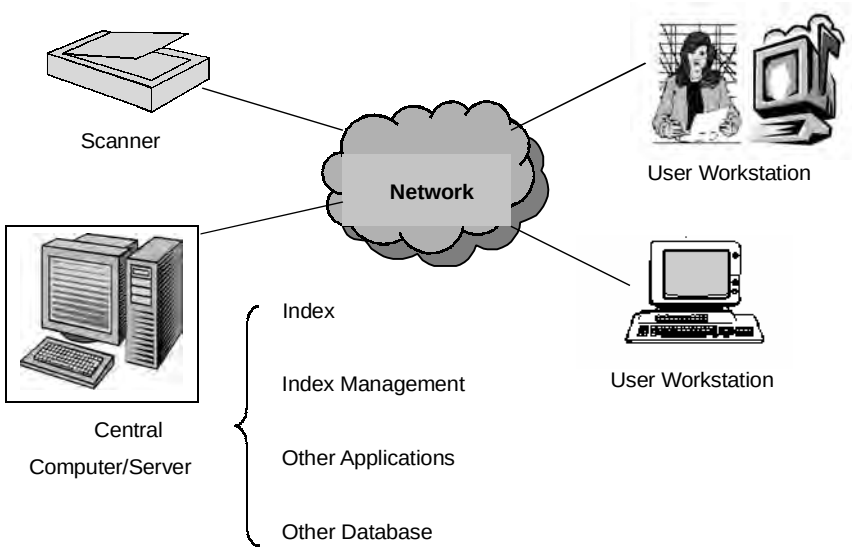


Fig. 9.3: Imaging System

Now, imaging is the method of getting digital documents from physical ones. An imaging system passes a paper document through a scanner that renders it digital and then stores the digital data as a bit mapped image of the document. Fig. 9.3 illustrates the imaging system. This image is stored on a permanent medium (magnetic tape, disk or optical disk). The keyword for each document that helps in indexing and retrieval are entered during scanning. The index usually is stored in a relational database on a high-speed magnetic disk. Access to stored image is always initiated by an index search. These image documents can be accessed by any workstation, which accesses the image server.

9.3.3 Point-of-Service

POS is one of the most widely installed examples of Client/Server technology, also known as Point of Sale. POS's represents the best model for the implementation of Client/Server applications by combining information technology, management information and trade processes on a single platform. For product distribution, inventory control, pricing, accounting, staff management, the POS's are used everywhere at the supermarket, hotel, restaurants, stores and auto-service stations. POS systems record each sale in a central database (server), using a scanner which reads the bar code on the products, so that retailers no longer have to wait for a periodic inventory check to find out what they need to reorder. Centralized buying ensures price through volume purchasing and efficient distribution chains.

Point-of-sale scanning is the starting point in the EDI chain that allows the retailer to track merchandise at the item level and provides detail information for demand forecasting. This way of managing inventory can eliminate the need to remark merchandise for discount and promotions to reduce inventory levels. POS systems feed data to automatic replenishment systems that constantly monitor inventory levels and trigger EDI transactions. These systems support smaller, more frequent deliveries, which improve in-stock positions and reduce on-hand inventory. Scanning is a valuable part of warehouse operations, as this expedites the rapid flow of goods through the distribution center by reducing manual receiving and checking procedures.

9.4 CLIENT/SERVER COMPUTING AND THE INTRANET

9.4.1 Intranet

Due to sheer numbers, the intranet will continue to be the main area of interest for Client/Server development. Intranet is a term used to refer to the implementation of Internet technologies within a corporate organization, rather than for external connection to the global Internet. Or in other words, the term Intranet refer to the whole range of internet base applications, including network news, gopher, web technology. An intranet-based approach to corporate computing includes the following advantages:

- Supports a range of distributed computing architecture (few central server or many distributed servers).
- Open architecture means large number of add-on applications available across many platforms.
- Can be implemented on virtually all platforms with complete interoperability.
- Rapid prototyping and deployment of new services.
- Structure to support integration of "legacy" information sources.
- Support a range of media types (Audio, video, interactive applications).
- Inexpensive to start, requires little investment either in new software or infrastructure.
- Virtually no training required on the part of users and little training required to developers, because the user services and user interfaces are familiar from the Internet.

9.4.2 Is the Intranet Killing Client/Server?

Some people think that there is so much interest in the intranet that Client/Server will fall by the wayside. Actually, the opposite is true. The intranet and Client/Server complement rather than compete. In fact, the migration to the intranet is actually a process of simply deploy Client/Server technology using the commodity Web technology. What the intranet brings to the table is a new platform, interface, and architectures. The intranet can employ existing Client/Server applications as true intranet applications, and integrate applications in the Web browser that would not normally work and play well together. The intranet also means that the vast amount of information on the Internet becomes available from the same application environment and the interface which is a valuable advantage. The intranet also puts fat client developers on a diet. Since most intranet applications are driven from the Web server, the application processing is moving off the client and back onto the server. This means that maintenance and application deployment become much easier, and developers don't have to deal with the integration hassles of traditional Client/Server (such as loading assorted middleware and protocol stacks).

There is a drawback to the intranet movement. The interest in the Web has taken most R&D funds away from traditional Client/Server. In many respects, the evolution of traditional Client/Server technology (middleware, databases, and front-end development tools) remained static while the technology vendors moved their products to the Web.

Moving to Proprietary Complexity: While the sheer simplicity of the intranet has driven its success, we are now in danger of driving this technology the same way we are driving client servers: to proprietary complexity. Where HTML and CGI were once commoly held standards, we now have proprietary APIs such as NSAPI and ISAPI that are pushing CGI aside. Java was considered the only way to program dynamic web applications, but now we have ActiveX as an alternative. Even Netscape now touts the standard HTTP as "legacy technology."

It's difficult at this time to determine if this movement to proprietary technology is a good thing. One thing for sure is that the simple architecture of just a few years ago is gradually disappearing. In many respects, intranet development is becoming as complex as Client/Server development. The trend is going to continue. We could see even more complexity in the world of Web development than we ever saw with Client/Server.

As interest in the Web shifts towards creating successful applications, we'll see more links between the intranet and traditional Client/Server. Right now, the tool vendors' fear that they will miss a large portion of the market puts them in a reactive rather than a proactive mode.

9.4.3 Extranet

Extranet is the similar concept to the intranet, using TCP/IP protocols and applications, especially the Web. The distinguished feature of the extranet is that it provides the access to corporate resources by outside clients (suppliers and customers of the organization). This outside access can be through the Internet or through other data communication networks. An extranet provides a simpler Web access and more extensive access to corporate resources, enforcing some security policies becomes a necessity. As with the intranet, the typical model of operation for the extranet is Client/Server. Some of the communication options available for operating intranet to outsiders to create an extranet.

- Long distance dialup access.
- Internet access to intranet with security.
- Internet access to an external server that duplicates some of a company's intranet data.
- Internet access to an external server that originate database queries to internal servers.
- Virtual private network.

9.5 FUTURE PERSPECTIVES

9.5.1 Job Security

Client/Server developers and application architects have a bright future. IDC (International Data Corporation) says that 33 per cent of organizations are already committed to Client/Server computing, despite the long break-in period. The Strategic Focus reports that three-tier Client/Server applications will grow from 4.9 per cent to 18.7 per cent in just two years. This growth will also fuel the rise of three-tier and n-tier Client/Server technology (such as TP monitors and distributed objects). Forrester Research estimates that the Client/Server market place will rise to $7.5 billion in 1996. Finally, the Gartner Group predicts that 65 per cent of all applications created in the next five years will employ Client/Server technology.

Clearly, the growth of Client/Server is steady, and as time goes on, the vast majority of applications we build for businesses will employ Client/Server technology. We still have a way to go, and many problems to solve, but it is certain that the Client/Server development game will always have room for talented people.

Look around today's "corporate downsized" job market. Those in Client/Server development have little reason to worry. There are more job openings than candidates, and that's been a pretty steady trend for several years.

9.5.2 Future Planning

Planning for the future is simply a matter of establishing the strategy and the tactics of our application development environment. From time to time, we need to consider the following:

- Re-evaluate our business objectives.
- Re-evaluate the current technology infrastructure.
- Determine the differences (what's missing?).
- Adjust our technology infrastructure to meet your objectives.

In most cases, this is a process of looking at new enabling technologies and paradigms that will better serve our needs. For example, if we need access to many applications for a single multi-platform user interface, then we may want to set a course for the intranet. Or, if we need to access a number of legacy resources as a single virtual system, then TP monitors and DCE are worth using. Of course, we would need to consider which enabling technology to employ, and then select the tools that support the enabling technology of choice.

While working out the plan for enabling technologies it is worth remembering that Vendors usually deliver 80 per cent of their promises up front, and the other 20 per cent shows up a few years later. For example, while Visual Basic and Power Builder offer proprietary application-partitioning mechanisms, they were so difficult to set up and so limited in what they could do that many development organizations abandoned them in their search for a quicker, easier way to move to n-tier. Visual Basic is fixing its limitations with DCOM and integration with Falcon and Transaction Server, but it took a few years.

It is always safer to follow the technology curve. While distributed objects were new and unproven a few years ago, they are proven today. DCOM is not yet proven, and proprietary application-partitioning tools such as Forte or Dynasty are almost there.

9.5.3 Conclusion

We can conclude some key features about client/server computing that are:

- Client/server distributes processing among computers on a network.
- In client/server environment most of the client provides GUI's facilities.
- Most of the servers provide SQL processing.
- Business rules can run either on the client or the server.
- Clients are typically programmed with a visual programming tool.

- Transaction processing is on of the biggest challenges of client/server computing.
- Middleware is the layer between clients and servers.
- Client/server standards are emerging.

In near future, cheap and powerful workstation technologies will be available to all with truly distributed applications using processing power where available, and providing information where required. Also information will be available for use to owners and authorized users without a constant need for system development by professional and their complex programming languages. The future will see information being captured at its source, and made available immediately to authorized users. Users will be able to avail information in original forms of data, such as image, video, audio, graphics, documents and spreadsheet without the need to be aware of various software requirements for information presentation. Successful organizations of the future, that those are market driven and competitive will be the ones that use client/server as an enabling technology to add recognized value to their products and services.

9.6 TRANSFORMATIONAL SYSTEM

The working environment of many of the organizations has been greatly affected by applications of Client/Server technologies. Following are the examples of technologies that have changed the trade processes.

(*i*) Electronic mail.

(*ii*) Client/server and user security.

(*iii*) Object oriented technology: CORBA.

(*iv*) Electronic data interchange.

9.6.1 Electronic Mail

Electronic mail is already the most heavily used network application in the corporate world. It is a facility that allows users at workstations and terminals to compose and exchange messages. However, the traditional e-mail is generally limited and inflexible. Intranet mail products provide standards, simple methods for attaching documents, sound, images, and other multimedia to mail messages.

The simplest form of electronic mail is the single system facility allowing the users of a shared computer system to exchange messages (see the Fig. 9.4 (*a*) given). The electronic mail facility is an application program available to any user logged onto the system. A user may invoke the electronic mail facility, prepare a message, and "send" it to any other user on the system. The act of sending simply involves putting the message in the recipient's mailbox. Mailbox is actually an entity maintained by the file management system and is in nature of a file directory. Any incoming mail is simply stored as a file under that user's mailbox directory. One mailbox is associated with each user.

With a single system electronic mail facility, messages can only be exchanged among users of that particular system. For a distributed mail system, a number of mail handlers (mail servers) connect over a network facility (e.g., WAN or internet) and exchange mail. That is illustrated in Fig. 9.4(b) given below:

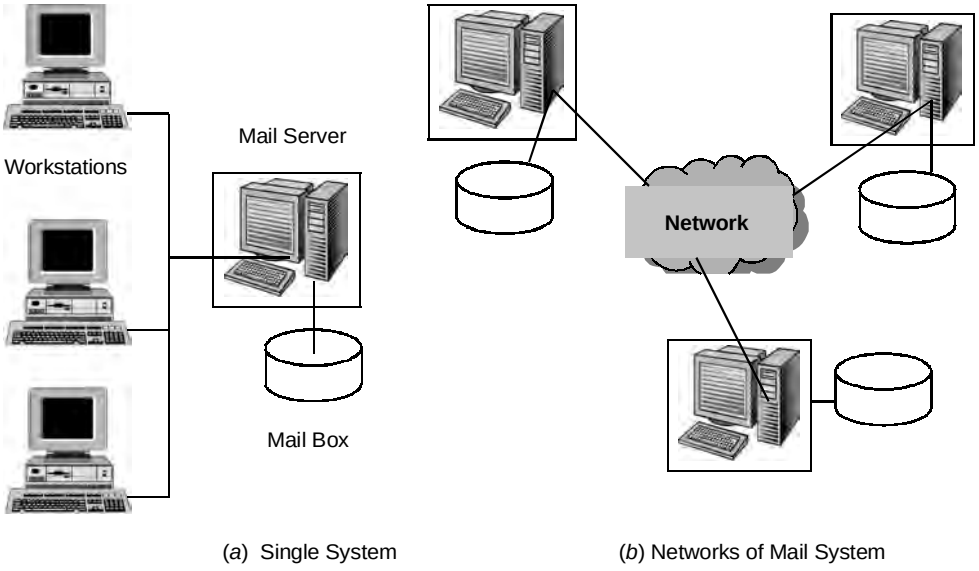


Fig. 9.4: Electronic Mail Configurations

Intranet mail system creates and manages an electronic mailing list that is an alias to multiple destinations. Mailing list is usually created to discuss specific topics. Any one interested in that topic may join that list, once a client has been added to the list. A user can ask question or respond to some one else's question by sending a message to the list address.

9.6.2 Client/Server and User Security

However, the very characteristic that make Client/Server popular are also what make it the most vulnerable to breaches in security. It is precisely the distribution of services between client and server that open them up to damage, fraud, and misuse. Security consideration must include the host systems, personal computers (PCs), Local Area Networks (LANs), Global Wide Area Networks (WANs), and users. Because security investments don't produce immediately visible returns and Client/Server buyers sometimes don't educate themselves about security, this area of development is often overlooked until a problem occurs.

Desktops are the front-end system devices, the ones that deal most directly with user input. They are also the least secure environments in Client/Server models. Clients connect to servers and these connections, if left open or not secured, provide entry points for

hackers and other intruders that may use data for nefarious purposes. Aside from physical client security in the form of disk drive locks or diskless workstations that prohibit the loading of unauthorized software or viruses, accessibility to all files stored on a workstation operating system is the other gaping security hole in clients. For example, the machine assumes that whoever turns on the computer is the owner of all the files stored on it. They even have access to configuration files. This could result in sabotage or the leaking of sensitive data. The transmission of corrupted data may also occur on the level of the operating system, outside the realm of Client/Server application security, as data is transferred to different tiers of the architecture.

However, the primary culprits of breaching client security are not hackers or viruses, but the users themselves. The front-line of defense in client security is user identification and authentication. The easiest way to gain illegal access to computers is to get users' login ID and passwords. Sometimes users pick short or easily guessed passwords or share their passwords with others. Password management provides a security measurement for this by requiring a minimum amount of characters to be used in passwords checking passwords for guess ability, and regularly asking users to change their passwords. For example, more organizations are adopting policies of 'pass phrases' rather than passwords that are more complicated and harder to identify or guess. The system contains a scheme (minimalist, multi-paradigm programming language) that proactively detects and blocks spyware. It also updates daily. Gateways are nodes on a network that create entrances to other networks. It routes traffic from workstations to broader networks. Therefore, securing the gateways will prevent malware from ever reaching the client.

Using Client/Server computing some of the secure systems can also be implemented, having a goal to provide secure services to clients with maximum possible performance. Emergency response vehicle can be quickly dispatched by dispatch operator to an incident and at the same time dealing can also be reported over the phone. This functionality can be provided 24 hours. It is now possible to duplicate all of the functionality of such an existing traditional design with the additional advantages of better performance, a Graphical User Interface (GUI), a single point of contact, higher reliability, and lower costs. With the help of a Client/Server-based system, the dispatch operator is empowered to oversee how staff and equipment are allocated to each incident. The operator uses a GUI to dynamically alter vehicle selection and routing. Maps may be displayed that show the location of all incidents, emergency response centers, and vehicles. Vehicles are tracked using Automatic Vehicle Locator (AVL) technology. Obstacles, such as traffic congestion, construction, and environmental damage (such as earthquakes) are shown on the map so the dispatcher can see potential problems at a glance. Workstation technology provides the dispatcher with a less stressful and more functional user interface. The dispatcher can respond quickly to changes in the environment and communicate this information immediately to the vehicle operator. The system is remarkably fault-tolerant. If a single workstation is operating, the dispatcher can continue to send emergency vehicles to the incident.

9.6.3 Object-oriented Technology: CORBA

As object oriented technology becomes more prevalent in operating system design; Client/ Server designers have begun to embrace this approach. Here client and servers ship messages back and forth between objects. Object communication may rely on an underlying message or Remote Procedure Call (RPC) structure or be developed directly on top of object oriented capabilities in the operating system. Clients that need a service send a request to an object request broker, which acts as a directory of all the remote services available on the network. The broker calls the appropriate object and passes along any relevant data. Then the remote object services the request and replies to the broker, which returns the response to the client. There are several object oriented approach for standardization of these object mechanism are COM (Common Object Model), OLE (Object Linking and Embedding), Common Object Request Broker Architecture (CORBA, see the Fig. 9.5).

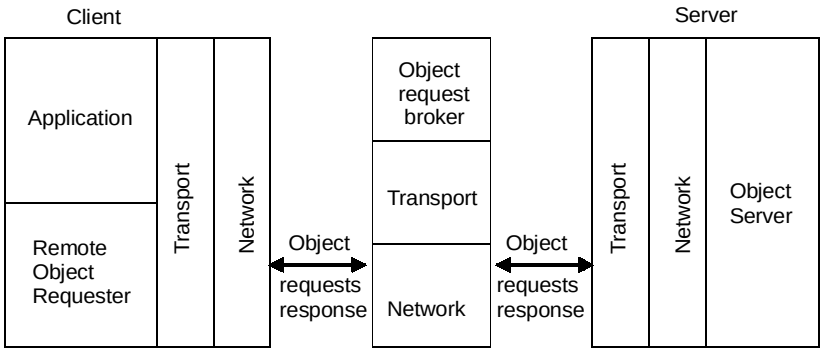


Fig. 9.5: Object Request Broker

An Overview of CORBA

The Object Management Group (OMG) was created in 1989. The OMG solicited input from all segments of the industry and eventually defined the CORBA standards.

CORBA specification has been implemented by numerous hardware and system software manufacturers, provides a rich and robust framework that operates across the heterogeneous computing platform. CORBA is a specification for an emerging technology known as distributed object management (DOM). DOM technology provides a higher level, object oriented interface on top of the basic distributed computing services.

At its most basic level, CORBA defines a standard framework from which an information system implementer or software developer can easily and quickly integrate network resident software modules and applications to create new, more powerful applications. It combines object technology with a Client/Server model to provide a uniform view of an enterprise computing system-everything on the network is an object. The highest level specification is referred to as the object management architecture (OMA), which addresses four architectural elements (ORB, CORBA services, CORBA facilities and CORBA domains are

also defined as a part of specifications. The term CORBA is often used to refer to the object request broker itself, as well as to the entire OMG architecture.

The role of ORB is to route request among the other architectural components. CORBA services, CORBA facilities and CORBA domains are also defined as a part of specifications. The key to integrating application object is the specification of standard interfaces using the interface definition language (IDL). Once all applications and data have an IDL-compliant interface, communication is independent of physical location, platform type, networking protocol, and programming language. An information system is created by using CORBA to mediate the flow of control and information among these software objects.

CORBA an Introduction

Mechanism that allows various clients to share/call the object (applications) over a mixed network, more specifically CORBA is a process of moving objects over network providing cross platform for data transfer.

CORBA compliance provides a high degree of portability. Within CORBA, objects are an identifiable entity which provides one or more services to clients. CORBA manages the identity and location of the objects, transport the request to the target object, and confirms that the request is carried out. A client that needs a service sends a request to an object request broker (which acts as a directory) of all the remote services available on the network, illustrated in Fig. 9.6.

The broker calls the appropriate object and passes along any relevant data, and then the remote object services the request and replies to the broker, which returns to the client. The object communication may rely on an underlying message or RPC structure or be developed directly on top of object-oriented capability in the operating system.

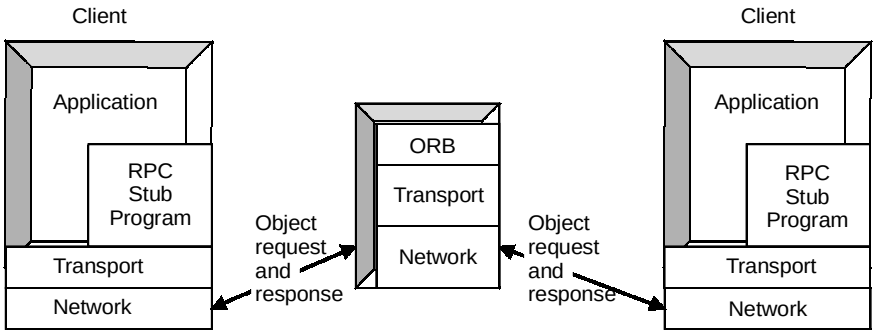


Fig. 9.6: *CORBA Remote Services*

CORBA Client and Servers

Like the Client/Server architecture, CORBA maintains the notions of Client and Server. In CORBA, a component can act as a client and as a server. Morever a component is considered a server if it contains CORBA objects whose services are accessible to other objects. Similarly, a component is considered a client if it accesses services from some other CORBA object.

Thus, a component can simultaneously provide and use various services, and so a component can be considered as a client or a server depending on the way.

More specifically, in CORBA application, any component that provides an implementation for an object is considered as a server, at least where that objects are concerned. If a component creates an object and provides others components with visibility to that object (or in other words, allows other components to obtain references to that object), that component acts as a server for that object; any requests made on that object by other components will be processed by the component that created the object. Being a CORBA server means that the component (the server) executes methods for a particular object on behalf of other components (the clients).

An application component can provide services to other application components while accessing services from other components. Here, the component is acting as a client of one component and as a server to the other components; see the Fig. 9.7 given below, illustrating those two components can simultaneously act as clients and servers to each other. In a CORBA application, the term client and server might depends on the context of the method being called and in which component that method's object resides. Although an application component can function as both a client and a server.

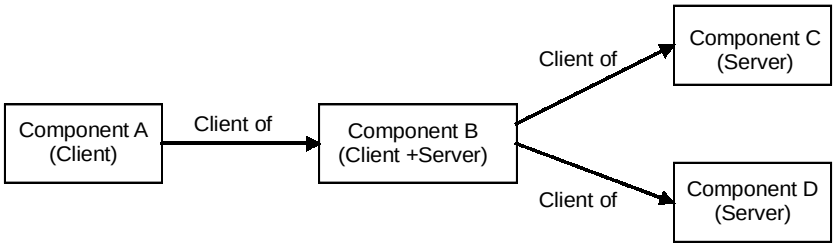


Fig. 9.7: Acting as a Client and a Server

CORBA Concepts

The basic idea is distributed computing, nowadays, most of the application are across the open environment, based on the connection of heterogeneous platforms. All modern business systems employ a network to connect a variety of computers, facilitating among applications. In the future, there will be continued evolution toward applications that exist as components across a network, which can be rapidly migrated and combined without significant effort. This is where CORBA shines, by providing unified access to applications, independent of the location of each application on network, also it provides:-

- Uniform access to services.
- Uniform discovery of resource/object.
- Uniform error handling methods.
- Uniform security policies.

These capabilities facilitate the integration and reuse of systems and system components, independent of network location and the details of underlying implementation technology. CORBA can be theoretically described based on the following three important concepts:

(*i*) Object-oriented model.

(*ii*) Open distributed computing environment.

(*iii*) Component integration and reuse.

(*i*) Object-oriented model: CORBA's object model is based on complete object approach in which a client sends a message to an object. The message identifies an object, and one or more parameters are included. The first parameter defines the operation to be performed, although the specific method used is determined by the receiving object. The CORBA object model comprises of:

Objects: An encapsulated entity that provides services to a client.

Request: An action created by a client directed to a target object that includes information on the operation to be performed and zero or more actual parameters.

Object creation and destruction: Based on the state of request, objects are created or deleted.

Types: An identifiable entity defined over values.

Interfaces: The specification of operations that a client can request from an object.

Operations: An identifiable entity that defines what a client can request from an object.

Object implementation in CORBA can be constituted in two parts, illustrated in the Fig. 9.8, first one is construction part and second one is execution part.

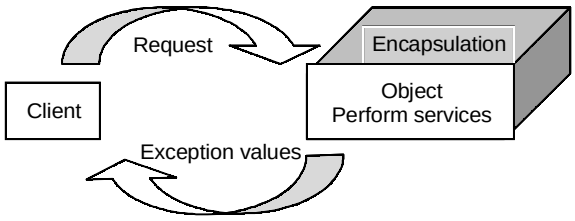


Fig. 9.8: Object Implementation

(*ii*) Open distributed computing environment: As we have earlier discussed that CORBA is based on a client server model of distributed computing. Within the Client/Server model, requests for services are made from one software component to another on a network. CORBA adds an additional dimension to this model by inserting a broker between the client and server components. The main objective of the broker is to reduce the complexity of implementing the interaction between the client and server. The broker plays two major roles. Primarily, it provides common services, including basic messaging and communication between client

and server, directory services, meta-data description, security services, and location transparency. Secondly, it insulates the application from the specifics of the system configuration, such as hardware platform and operating system, network protocol, and implementation languages.

(*iii*) **Component integration and reuse:** The integration is the combination of two or more existing components. With a good integration techniques and tools, reuse can be achieved up to significant degree. Broker defines custom interfaces for each interaction between components. Each interface is defined just once and subsequent interactions are handled by the broker. With CORBA IDL, these interfaces can be defined in a standardized, platform independent fashion.

9.6.4 Electronic Data Interchange

Electronic data Interchanged uses direct link between computers, even computers on different sites, to transmit data to eliminate data sent in printed form. It is a controlled transfer of data between business and organizations via established security standards. One of the examples of EDI is shown in Fig. 9.9.

EDI is generally thought of as replacing standardized documents such as order forms, purchase orders, delivery notes, receiving advices and invoices in a standardized electronic message format. EDI documents are electronically exchanged over communication networks which connect trading partners to one another. These documents are stored in user mailboxes on the networks' EDI server from where they can be downloaded/uploaded at the user is convenience from any one of the workstations. But it differs from electronic mail in that it transmits an actual structured transaction (with field such as the transition date, transaction amount, senders name and recipient name) as opposed to unstructured text message such as a letter.

The main purpose of EDI is cost reduction by eliminating paper document handling. Faster electronic document transmission further saves time and man power by avoiding the need to re-key data. And the data arrival rate is much faster that mail.

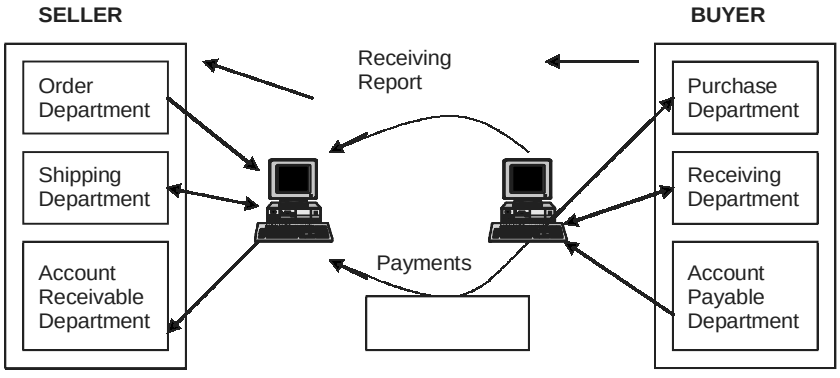


Fig. 9.9: EDI as an Example

EXERCISE 9

1. What is the future of Client/Server computing in the following technologies:-
   (*i*) Geographic Information System (GIS).
   (*ii*) Point of Service Technology (POS).
   (*iii*) Electronic Data Interface Technology (EDI).
   (*iv*) Multimedia.
2. What is the future of Client/Server computing in the following technologies?
   (*i*) Electronic Document Management.
   (*ii*) Full Text Retrieval.
   (*iii*) Geographic Information System.
3. What are different enabling technologies? Explain Expert System, Imaging and Electronic Document Management.
4. Discuss the changing role of Server's to provide the balance computing in Client/Server environment.

This page intentionally left blank

References

Alapali, Sam. R. *Expert Oracle Database 10g Administration*, A Press Burkeley, USA.

Bhattacharjee, Satyapriya. *A Textbook of Client/Server Computing*.

Comer, Douglas. E. *Computer Networks and Internets*.

Comer, Douglas. E. *Internetworking with TCP/IP*.

Coronel, R. *Database System,* Pearson Education, New Delhi.

Coulouris, George, et al. *Distributed System : Concepts and Design*

Crowley, Charles. *Operating System a Design Oriented Approach*, Tata McGraw-Hill Pub., New Delhi.

CSI Communications, *A Comparative Study on 2-tier and 3- tier Client/Server Architecture*, September, 2001.

CSI Communications, *Client/Server Technology, A Polymorphic Visage*, June, 2002.

Date, C. J. *An Introduction to Database Systems*, Pearson Education, New Delhi.

Desai, Bipin C. *An Introduction to Database Systems*, Galgotia Pub., New Delhi.

Dhamdhere, D. M. *Operating Systems* , Tata McGraw-Hill Pub., New Delhi.

Edward and Jerry. *3-Tier Client/Server at Work*.

Elmarsi, Ramez and Navathe, Shamkant. B. *Fundamentals of Database System*.

Eversest, Gordon C. *Database Management*, Tata McGraw-Hill Pub., New Delhi.

Forouzan, Behrouz A. *Data Communication and Networking*, Tata McGraw-Hill Pub., New Delhi.

Gallo, Michael. A. and Hancock, William. M. *Computer Communications and Networking Technology*, Books/Cole Pub. Company, Singapore.

Godbole, Achynt S. *Operating Systems*, Tata McGraw-Hill Pub., New Delhi.

Greenwald, Rick. et al. *Oracle Essantials*.

Haecke, Bernard Van. *Java-Database Connectivity*.

Hahn, Harley. *The Internet Complete Reference*, Tata McGraw-Hill Pub., New Delhi.

Halsall, Fred and Kulkarni, Lingana. *Computer Networking and the Internet*, Pearson Education, New Delhi.

Halsall, Fred. *Data Communication, Computer Networks & Open Systems*.

Harvey, Dennis and Beitler, Steve. *The Developer's Guide to Oracle Web Application* Server 3.

Hutehinson, Sarah, E. and Sawyer, Stcey. C. *Computer Communications, Information*,Tata McGraw-Hill Pub., New Delhi.

Jani, Madhulika and Jain , Satish. *Data Communication and Networking*, BPB Publications, New Delhi.

Kalakota, Ravi and Whinston, Andrew B. *Frontiers of Electronic Commerce*.

Kurose, James. F. and Ross, Keith. W. *Computer Networking*, Pearson Education, New Delhi.

Leon Alexis & Leon Mathews. *Data Base Management System*.

Majumdar, Arun K. and Bhattacharyya, Pritimoy. *Database Management Systems*, Tata McGraw-Hill Pub., New Delhi.

Martin, James. *Principles of Database Management*, PHI, New Delhi.

Milenkovic, Milan. *Operating System: Concepts and Design*, Tata McGraw-Hill Pub., New Delhi.

Miller, Michael A. *Data and Network Communications*, Thomson Asia Pvt. Ltd, Singapore.

Nutt, Gary. *Operating System: A Modern Perspective*, PHI, New Delhi.

Powell, Gavin. *Beginning Database Design*, Wiley Publishing Inc., USA.

Prabhu C. S. R. *Object-oriented Database Systems*, PHI, New Delhi.

Ramakrishnan, Raghu and Gehrke, Johannes. *Database Management Systems*, McGraw-Hill, Boston.

Ritchie, Colin. *Operating Systems in Corporating Unix and Windows*, PBP Publication, New Delhi.

Rosenberger, Jeremy.: *Teach Yourself CORBA,* Techmedia, New Delhi.

Scqnk, Jeffrey D. *Novell's Guide to C/S Design and Implementation*.

Silberchatz, Abraham, et al. *Operating System Principles*, John Wiley & Sons, Singapore.

Silberchatz, Abraham. et al. *Database System Concepts*.

Sinha, Pradeep K. *Distributed Operating System Concepts and Design*, PHI Pub., New Delhi.

Smith, Pattric N. and Ganguarich, Evan. *Client Server Computing*, PHI, New Delhi.

Perry, James T. and Laler Joseph G. *Understanding ORACLE.*

Sperik, Mark and Sledge, Orryn. *SQL-Server 7.0 DBA Survival Guide*.

Stalling, William. *Business Data Communication*.

Stalling, William. *Operating Systems*, PHI, New Delhi.

Tanenbaum, Andrew S. and Woodhull, Albert S. *Modern Operating System*, PHI, New Delhi.

Tanenbaum, Andrew S. and Woodhull, Albert S. *The MINIX book Operating System Design and Implementation*, Pearson Education, New Delhi.

Tanenbaum, Andrew S. *Computer Networks*, Pearson Education, New Delhi.

Shay, William A. *Understanding Data Communications and Networks*, Books/Cole Pub. Company, Singapore.

Thomas, Robert. M. *Introduction to Local Area Network*.

Travis, Dawna D. *Client/Server Computing.*

Vaskevitch, David. *Client/Server Unleashed*.

URLS

http://www.ssuet.edu.pk/taimoor/books/0-672-30473-2/csc09.htm
http:/www.ssuet.edu.pk/taimoor/books/0-672-30473-2/index.htm
http:/www.corba.ch/e/3tier.html
http:/www.acs.ncsu.edu/~nacsrjm/cs_stnd/cs_stnd.html
http://www-bfs.ucsd.edu/systems/cs/standrd.htm
http://www.sei.cmu.edu/str/descriptions/clientserver_body.html
http://www.softis.is
http://www.dciexpo.com/geos/
http://www.byte.com/art/9504/sec11/art4.htm
http://www.iterrasoft.com/ClientServer.htm
http://www.dpu.se/CTR/ctrcli_e.htm
http://www.tietovayla.fi/borland/techlib/delvpowr.html
http://www.opengroup.org/dce/successes/case_kredbank.htm
http://linuxtoday.com/developer/2001120600920OSSW
http://www.freeos.com/articles/2531/
http://www.linuxgazette.com/issue68/swieskowski.html
http://www.hrmanagement.gc.ca/gol/learning/interface.nsf/engdocBasic/1.html
http://www.conceptsystems.com
http://www-staff.it.uts.edu.au/~chin/dbms/cs.htm
http://www.exforsys.com
http://www.testingcenter.com/oviewtc.html
http://www.education-online earch.com
http://orca.st.usm.edu/~seyfarth/network_pgm/net-6-3-3.html
http://www.se.cuhk.edu.hk
http://www.gerrardconsulting.com/GUI/TestGui.html

This page intentionally left blank

Index

A

A graphical User Interface (GUI) 47
Application Processor (AP) 75-76
Application server 2, 17
Application services 97
Application-programming interface 70
Asynchronous Transfer Mode (ATM) 133

B

Balanced computing 171
Banyan VINES 130
Browser/server models 157
Business information system 55

C

Client/server application 79, 80
Client/server computing 1
Client-based processing 85
Client 3
Clients/server 19
Communication services 48
Cooperative processing 85
CORBA 4, 17, 188
CORBA an introduction 189
CORBA client and servers 189
CORBA's object model 191
Crypto-capable routers 116

D

Data distribution 72
Data Terminal Equipment (DTE) 134
Database centered systems 81
Database middleware component 70
Database Processor (DP) 75
Database server 3, 48
Database services 48, 95, 98
Database translator 70
Decision-Support Systems (DSS) 81
Development of client/server systems 29
Digital pen 109
Direct communication 88
Directory services server 3
Distributed computing 8
Distributed database 59
Distributed DBMS 74
Distributed objects 83
Downsizing 38

E

Electronic mail (E-mail) 83
Enterprise computing 28
Event handler 155
Event-driven 155

F

Fat client 19
Fat client 85
Fat clients 4
Fat server 19
Fat servers 4
Fax server 2
Fax services 48
FDDI (Fiber Distributed Data Interface) 135
File server 2
File services 48, 98
File sharing 81
File transfer protocol 158
First-tier (client-tier) 13

G

Groupware 82
Groupware servers 3
Groupware services 48

H

Host-based processing 84
Hybrid architecture 69

I

IBM LAN server 130
Indirect communication 88
Internet protocols 157
Interrupt handler 155

L

LAN manager 129
Light pen 108

M

Magnetic disks 111
Magnetic tape 110
Mail server 2
Marshalling 90
Memory leaks 155
Message flooding 34
Message passing 87
Message server 16
Middleware 44, 178
Middleware 44
middleware components 52
Miscellaneous services 49
Misconceptions 22
Multimedia Document Managements (MMDM) 82
Multi-threaded architecture 68
MVS 126

N

NetWare 125
Network File System (NFS) 136
Network interface card 131
Network management 28
Network translator 71
Network transparency 107
Notebook computers 109
Novell NetWare 128
N-Tier 9

O

Object application servers 4
Object application services 49
Online 3
Online transaction processing 3, 82
Optical disk 112
ORB 17
ORB architecture 17
OS/2 125
Overview of CORBA 188

P

Packet replay 34
Pen drives 114
Performance testing 152
Print server 2
Print services 48
Print/fax services 95
Processors and servers 177
Process-per-client architecture 68

R

Remote database 58
Remote procedures call 88
Remote user interface 58
Rightsizing 38, 39
Risk driven testing 151
RMI system 166

S

Second-tier (application-server-tier) 13
Service overloading 34
Simple Mail Transfer Protocol (SMTP) 136
Simple Network Management Protocol (SNMP) 135-36
Stateful server 5
Stateless server 4
Stateless vs stateful servers 5

T

Tape drives 114
Terminal server 116
Thin 3
Thin client 4
Three-tier 9
Three-tier (database-server-tier) 14
Transaction processing monitors 15
Transaction servers 3
Transaction services 48
Transactional processing 83
Transaction-processing monitors 178
Two-tier 9

U

UNIX 127
UNIX workstations 105
UPS (Uninterruptible Power Supply) 118
Upsizing 39

V

Virtual groups 142
Virtual private networks 116
VMS 127
Voltage sag 118
Voltage spike 119

W

Web application services 49
Web applications 168, 170
Web client 160
Web server 3
Web services 158
Windows NT 125
Wired Equivalent Privacy (WEP) 117
Wireless Network Protection 117

X

X-server 106, 107
X-terminal 106
X-terminals 91
X-window system 105

Z

Zip drives 114