COMPLETE BOOK IN PRINT & ON CD-ROM

ENCYCLOPEDIA OF

GRAPHICS FILE FORMATS

SECOND EDITION



THE COMPLETE REFERENCE ON CD-ROM
WITH LINKS TO INTERNET RESOURCES



FOR PC, MACINTOSH, AND UNIX PLATFORMS



JAMES D. MURRAY AND WILLIAM VANRYPER

O'REILLY & ASSOCIATES, INC.

ENCYCLOPEDIA OF

GRAPHICS FILE FORMATS

ENCYCLOPEDIA OF

GRAPHICS FILE FORMATS

SECOND EDITION

James D. Murray and William vanRyper

O'REILLY & ASSOCIATES, INC.
BONN • CAMBRIDGE • PARIS • SEBASTOPOL • TOKYO

Encyclopedia of Graphics File Formats, Second Edition
by James D. Murray and William vanRyper

Copyright © 1996, 1994 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 103 Morris Street, Suite A, Sebastopol, CA 95472

Editor: Deborah Russell

Production Editor: David Futato

Software Development: Norman Walsh & Ken DeCanio

Printing History:

July 1994:	First Edition
April 1996:	Second Edition

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

Specific copyright notices and restrictions for specification documents and programs included on the CD-ROM accompanying this book are included on that CD-ROM.

All of the specification documents and programs described in this book and provided by vendors for inclusion on the CD-ROM are subject to change without notice by those vendors.

O'Reilly & Associates, Inc., and the authors have used their best efforts in preparing this book, the CD-ROM accompanying this book, and the programs and data contained therein. However, the authors and publisher make no warranties of any kind, expressed or implied, with regard to the documentation or programs or data contained in this book or the CD-ROM. O'Reilly & Associates, Inc., and the authors specifically disclaim, without any limitation, any implied warranties of merchantability and fitness for a particular purpose with respect to the CD-ROM, the programs therein and/or techniques described in the book. In no event shall the authors or publishers be responsible or liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential or any other damages in connection with or arising out of furnishing, performance, or use of this book or the programs or data.



This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-161-5 [9/96]

TABLE OF CONTENTS

Preface ..... *xv*

PART ONE

Overview

1. Introduction **3**

The Basics 4
Graphics Files 5
Graphics Data 7
Types of Graphics File Formats 12
Elements of a Graphics File 20
Converting Formats 22
Compressing Data 22
Format Summary 24

2. Computer Graphics Basics **27**

Pixels and Coordinates 27
Pixel Data and Palettes 31
Color 45
Overlays and Transparency 51
For Further Information 52

3. Bitmap Files **55**

How Bitmap Files Are Organized 55
Header 57
Examples of Bitmap Headers 62
Bitmap Data 65
Footer 72

Ex.1007 / Page 5 of 1158
APPLE INC.

Other Bitmap File Data Structures 73
Other Bitmap File Features 73
Pros and Cons of Bitmap File Formats 74

4. Vector Files **75**

Vector Versus Bitmap Files 75
What Is Vector Data? 75
Vector Files and Device Independence 76
Sources of Vector Format Files 77
How Vector Files Are Organized 77
Vector File Size Issues 82
Scaling Vector Files 83
Text in Vector Files 83
Pros and Cons of Vector Files 84

5. Metafiles **87**

Platform Independence? 88
How Metafiles Are Organized 88
Pros and Cons of Metafiles 88

6. Platform Dependencies **91**

Byte Order 91
File Size and Memory Limitations 93
Floating-Point Formats 94
Bit Order 94
Filenames 97
For Further Information 99

7. Format Conversion **101**

Is It Really Possible? 101
Don't Do It If You Don't Need to 102
... But If You Do 102
Other Format Conversion Considerations 106

8. Working With Graphics Files **107**

Reading Graphics Data 107
Writing Graphics Data 117
Test Files 120
Corruption of Graphics Files 121

Ex.1007 / Page 6 of 1158
APPLE INC.

Encryption of Graphics Files 127
Viruses in Graphics Files 137
Designing Your Own Format 139
Writing a File Format Specification 141
Trademarks, Patents, and Copyrights 144

9. Data Compression **153**

Data Compression Terminology 155
Pixel Packing 158
Run-Length Encoding (RLE) 160
Lempel-Ziv-Welch (LZW) Compression 170
CCITT (Huffman) Encoding 179
JPEG Compression 191
JBIG Compression 205
ART Compression 210
Fractal Image Compression 212
For Further Information About Data Compression 217

10. Multimedia **219**

Beyond Traditional Graphics File Formats 219
Multimedia File Formats 220
Types of Data 221
For Further Information 230

PART TWO

Graphics File Formats

Adobe Illustrator 235
Adobe Photoshop 250
Atari ST Graphics Formats 261
AutoCAD DXF 276
Autodesk 3D Studio 282
BDF 310
BRL-CAD 315
BUFR 318
CALS Raster 321
CGM 330
CMU Formats 336
DKB 342
Dore Raster File Format 344

Ex.1007 / Page 7 of 1158
APPLE INC.

DPX 350
Dr. Halo 368
DVM Movie 372
Encapsulated PostScript 376
FaceSaver 386
FAX Formats 389
FITS 392
FLI 401
GEM Raster 417
GEM VDI 423
GIF 429
GRASP 451
GRIB 457
Harvard Graphics 460
Hierarchical Data Format 462
IFF 465
IGES 484
Inset PIX 487
Intel DVI 497
JPEG File Interchange Format 510
Kodak Photo CD 516
Kodak YCC 520
Lotus DIF 522
Lotus PIC 529
Lumena Paint 532
Macintosh Paint 537
Macintosh PICT 544
Microsoft Paint 549
Microsoft RIFF 554
Microsoft RTF 564
Microsoft SYLK 569
Microsoft Windows Bitmap 572
Microsoft Windows Metafile 592
MIFF 599
MPEG 604
MTV 613
NAPLPS 617
NFF 620
OFF 627
OS/2 Bitmap 630
P3D 651
PBM, PGM, PNM, and PPM 658

Ex.1007 / Page 8 of 1158
APPLE INC.

PCX 662
PDS 678
Pictor PC Paint 683
Pixar RIB 695
Plot-10 698
PNG 700
POV 720
Presentation Manager Metafile 725
PRT 730
QRT 736
QuickTime 747
Radiance 760
Rayshade 764
RIX 772
RTrace 776
SAF 786
Sense8 NFF 796
SGI Image File Format 801
SGI Inventor 806
SGI YAODL 812
SGO 818
SPIFF 822
Sun Icon 838
Sun Raster 841
TDDD 846
TGA 860
TIFF 880
TTDDD 909
uRay 914
Utah RLE 917
VICAR2 921
VIFF 925
VIS-5D 935
Vivid and Bob 943
Wavefront OBJ 946
Wavefront RLA 953
WordPerfect Graphics Metafile 964
XBM 983
XPM 987
XWD 992
ZBR 998

Ex.1007 / Page 9 of 1158
APPLE INC.

PART THREE

Appendices

Appendix A: Graphics Files and Resources on the Internet **1005**

Encoding of Graphics Files 1005
Email 1010
USENET News 1011
Mailing Lists 1017
FTP Archives 1018
Archie 1019
The World Wide Web (WWW) 1020
Internet Graphics Resources 1029
For Further Information 1039

Appendix B: Graphics Files and Resources on the Commercial Services **1041**

CompuServe 1041
America Online 1043
Bulletin Board Systems 1044

Appendix C: Installation and Setup **1047**

Using the CD-ROM 1048
Installing GFF 1050
Removing GFF 1051
Which Browsers Can I Use? 1052
Accessing Software on the CD-ROM 1053
How Does GFF Work? 1053
Customizing GFF 1054

Glossary *1061*
Index *1099*

Ex.1007 / Page 10 of 1158
APPLE INC.

List of Figures

1-1 The graphics production pipeline 6
1-2 Vector data 8
1-3 Bitmap data 9
1-4 Object data 11

2-1 Physical and logical pixels 28
2-2 Displaying data with few colors on a device with many colors 31
2-3 Displaying data with many colors on a device with few colors 31
2-4 Using a palette to specify a color 33
2-5 Types of palettes 40

3-1 Organization of pixel data into scan lines (24-bit image) 68
3-2 Organization of pixel data into color planes 69
3-3 Examples of bitmap data organization (contiguous scan lines, strips, and tiles) 70

4-1 Gradient fill 81

9-1 Pixel packing 159
9-2 Run-length encoding variants 162
9-3 Basic run-length encoding flow 165
9-4 Bit-, byte-, and pixel-level RLE schemes 167
9-5 RLE scheme with three bytes 168
9-6 RLE scheme with 1- and 2-byte vertical replication packets 169
9-7 CCITT Group 3 encoding 183
9-8 CCITT Group 3 encoding (EOL code) 184
9-9 CCITT Group 3 encoding (RTC code) 185
9-10 Group 3 CCITT encoding (TIFF Compression Type 2) 186
9-11 JPEG compression and decompression 195

DPX-1 DPX format 352
DPX-2 Packed bits 365

Ex.1007 / Page 11 of 1158
APPLE INC.

GIF-1 GIF87a file layout 433
GIF-2 Arrangement of interlaced and non-interlaced scan lines 439
GIF-3 Layout of a GIF89a file 443

IFF-1 A chunk file structure 467
IFF-2 FORM ILBM file structure 468
IFF-3 FORM 8SVX file structure 468
IFF-4 CAT ILBM file structure 469
IFF-5 CAT JJJJ file structure 469
IFF-6 LIST ILBM file structure 470

OS/2 Bitmap-1 Monochrome/color data format 641
OS/2 Bitmap-2 Multi-image data format 643

SPIFF-1 Format of a SPIFF directory entry 829

TGA-1 Run-length encoding packet types 870
TGA-2 Pixel data formats 871

TIFF-1 Three possible physical arrangements of data in a TIFF file 884
TIFF-2 Logical organization of a TIFF file 885
TIFF-3 Format of an Image File Directory 888

Ex.1007 / Page 12 of 1158
APPLE INC.

LIST OF TABLES

1-1 Graphics File Formats Described in This Book 24

2-1 Equivalent RGB, CMY, and HSV values 48

Autodesk 3D Studio-1 Autodesk 3D Studio Chunks 284

CALS Raster-1 Typical CALS Raster Pel Count Values 327

GEM VDI-1 GEM Metafile Items and Commands 425

PNG-1 PNG Chunks 704

SAF-1 Generic SAF Tags 787
SAF-2 Additional Tags for SAF Image Data 790
SAF-3 Tags for Ordered Pair Data 792

SPIFF-1 Standard SPIFF Directory Entries 830

TGA-1 TGA Palette Entry Sizes 865

TIFF-1 TIFF Tag Types Listed Alphabetically by Name 890
TIFF-2 Minimum Required Tags for Each TIFF Class 894

A-1 Netscape MIME Types (for X Windows) 1026

Ex.1007 / Page 13 of 1158
APPLE INC.

PREFACE

Why did we write this book? The short answer is that *graphics file formats are immortal.* Like it or not, data files from the dawn of the computer age are still with us, and they're going to be around for a long time to come. Even when the way we think about data itself changes (as it inevitably will), hundreds of millions of files will still be out there in backup storage. We'll always need a way to read, understand, and display them.

Computer technology evolves rapidly. Hardware, particularly on the desktop, turns over every year or so. Software can become obsolete overnight with the release of a new version. The one thing that remains static is data, which for our purposes means information stored in data files on disk or tape. In this book we're interested in one specific type of data—that used for the interchange and reconstruction of graphics images.

Graphics data files are structured according to specific format conventions, which are (or should be) recorded in format specification documents written and maintained by the creator of the format. Not all formats are documented, however, and some documents are so sparse, poorly written, or out of date that they are essentially useless. Moreover, some format specifications are very difficult to obtain: the creator of the format might have moved; the format might have been sold to another organization; or the organization that owns the format might not actively support or distribute it. These facts make it difficult for someone who needs to find out about the specifics of a particular graphics file format to locate and understand the file format specification. We wrote this book because we saw a need for a centralized source of information, independent of the commercial marketplace, where anyone could obtain the information needed to read graphics files.

Ex.1007 / Page 14 of 1158
APPLE INC.

When we set out to write this book, we asked the obvious questions: How would we implement an existing format? What resources would we need? Ideally, we would like to have on hand a good book on the subject, and perhaps some working code. Barring that, we'd make do with the original format specification and some advice. Barring that, we'd scrape by with the format specification alone. This book provides as much of this as possible; the format specification is here in most cases, as is some code—even some advice, which, because it's coming from a book, you're free to take or leave as you choose.

To give you some idea about what was on our minds during the planning of this book, we'd like to mention some issues that frequently come up for programmers who need to learn about and implement file formats. In the course of writing this book, both of us (as consultants and veteran users of networked news and bulletin board systems) talked with and observed literally hundreds of other programmers. The following is a sampling of questions frequently asked on the subject of graphics file formats, and comments on how we have addressed them in this book:

"How can I get a copy of the written specification for format XYZ?"

Rarely does a day go by without a request for a written format specification—TIFF, GIF, FaceSaver, PNG, QRT, and many, many more. Unfortunately, there is no single source for even the most common format specifications. A number of format archives are available online, but they contain only what the maintainer has the time and resources to assemble. Each of the books previously on the market has offered a limited subset of the specifications out there.

"I'm trying to implement specification XYZ. I'm having trouble with ABC."

Programmers almost always believe that only the specification document is needed in order to implement a file format. Sadly, if you read a few format specifications, you'll soon discover that there is no law requiring that documentation be written clearly. Specifications, like all technical documents, are written by people with varying degrees of literacy, knowledge, and understanding of the subject in question. Consequently, they range from clearly written and very helpful to unorganized and confusing. Some documents, in fact, are nearly useless. The programmer is eventually forced to become conversant with the oral tradition.

Even if the specification document is well done, written between the lines is a complex set of assumptions about the data. How complicated can a format be? After hours of fiddling with color triples, index map offsets, page tables, multiple header versions, byte-order problems, and just plain bad design, you may

Ex.1007 / Page 15 of 1158
APPLE INC.

well find yourself begging for help while the clock counting your online dollars ticks on. Another goal of this book is to provide a second opinion for use by programmers who find themselves confused by the contents of the documents.

"What does Z mean?"

In this case, Z is basic technical graphics information. Everything a programmer needs to know to read, write, encode, and decode a format is in the specification document, right? Unfortunately, writers of format specifications often use vocabulary foreign to most programmers. For instance, the format might have been created in support of an application that used terminology from the profession of the target users. The meaning of a term might have changed since the time that the format was written, years ago. You might also find that different format specifications have different names for the same thing (e.g., color table, color map, color palette, look-up table, color index table, and so on). In this book, we provide basic guidance whenever possible.

"What is an X.Y file?"

If you scan the computer graphics section of any online service, bulletin board system, or news feed, you will find numerous general questions from users about graphics files, the pros and cons of each format, and sources of image files. Surprisingly, there is no single source of information on the origin, use, and description of most of the graphics file formats available today. Some of this information, particularly on the more common formats (e.g., TIFF, GIF, PCX), is scattered through books and magazine articles published over the last ten years. Other information on the less common formats is available only from other programmers, or (in some extreme cases) from the inventor of the format. Another goal of this book is to include historical and contextual information, including discussions of the strengths and weaknesses of each format.

"Is there a newer version of the XYZ specification than version 1.0?"

Occasionally, this question comes from someone who, specification in hand, just finished writing a format reader only to have it fail when processing sample files that are known to be good. The hapless programmer no doubt found a copy of the format specification, but not, of course, the latest revision. Another of our goals is to provide access to the latest format revisions in this book and keep this information up to date.

"How can I convert an ABC file to an XYZ file?"

Programmers and graphic designers alike are often stumped by this question. They've received a file from a colleague, an author, or a client, and they need to read it, print it, or incorporate it in a document. They need to convert it to

Ex.1007 / Page 16 of 1158
APPLE INC.

something their platform, application, or production environment knows how to deal with. If this is your problem, you'll find this book helpful in a number of ways. In the first place, it will give you the information you need to identify what this file is and what its conversion problems are. We'll give you specific suggestions on how to go about converting the file. Most importantly, on the CD-ROM that accompanies this book, we've included a number of software packages that will convert most graphics files from one format to another. Whether you are operating in an Windows, MS-DOS, OS/2, Macintosh, or UNIX environment, you should be able to find a helpful tool.

About This Book and the CD-ROM

We'd like to make it easier for you to understand and implement the graphics file formats mentioned in this book. Where does information on the hundreds of graphics file formats in use today come from? Basically, from four sources:

- Format specifications. These should be the ultimate references, shouldn't they? Unfortunately specifications aren't always available (or useful!). Nevertheless, they are the starting point for information about file formats.
- Secondary sources (magazine articles, books). These are most useful when the specification isn't handy and the author can provide some kind of insight or relevant experience.
- Sample code. This is all we usually want, isn't it? Unfortunately the sample code may not work right, may be out of date, or may be too platform-specific for your present needs.
- Sample images. Images fully conforming to the format specification might not seem like a source of information until you actually need something on which to test your application.

What we've tried to do is to collect these four elements together in one place. Of course not all were available for every format, and sometimes we weren't allowed to include the original specification document on the CD-ROM that accompanies this book. Nevertheless, we've pulled together all the information available. Taken together, the information provided in this book and in the materials on the CD-ROM should allow you to understand and implement most of the formats. In this second edition—more about this later—we also provide links on the CD-ROM to the O'Reilly GFF Web Center on the World Wide Web, where we're able to provide up-to-date information and additional resources, as they become available.

Ex.1007 / Page 17 of 1158
APPLE INC.

Our primary goal in writing this book is to establish a central repository of graphics file format specifications. Because the collected specification documents (not to mention the sample images and associated code and software packages!) total in the hundreds of megabytes, the best way to put them in your hands is on a CD-ROM. What this means is that the CD-ROM is an integral part of the book, if only for the fact that all this information could never be crammed between two covers.

We've written an article describing each graphics file format; this article condenses and summarizes the information we've been able to collect. In some cases this information is extensive; in other cases it's not much. This is the name of the game, unfortunately. When we do have adequate information, we've concentrated on conveying some understanding of the formats, which in many cases means going through them in some detail. Remember, though, that sometimes the specification document does a better job than we could ever do of explaining the nitty-gritty details of the format.

On the CD-ROM, you'll find the original format specifications (when available and when the vendors gave us permission to include them). If we know how to get the specifications, but couldn't enlist the aid of the vendors, we tell you where to go to find them yourself. Also on the CD-ROM is sample code that reads and writes a variety of file formats, and a number of widely-used third-party utilities for file manipulation and conversion. Finally, we've included sample images for many formats. If you have Internet access, you'll be able to get updates and new resources at our Web site.

About the Online Product

The first edition of this book was a traditional text book, accompanied by a CD-ROM that contained all of the information mentioned in the preceding section. In the second edition, we've added to the book and completely retooled the CD-ROM. Instead of finding a set of resource files there, you'll find a software product that lets you browse the complete text of the book (using Enhanced Mosaic, also provided) and all of the specs, images, code examples, and software packages we've been able to pull together, plus (with an Internet connection) a link to the GFF Web Center.

GFF Web Center Online

http://www.ora.com/centers/gff/

Ex.1007 / Page 18 of 1158
APPLE INC.

Follow the installation instructions (see Appendix C) to get the product up and running. From the Main Menu, you'll have the following choices:



- **Formats.** Information about all of the formats we describe in Part Two of this book. From the format pages you'll be able to link to vendor specification documents describing the format, as well as to appropriate images and code examples and to software packages that let you view, convert, and otherwise manipulate the format. From Formats, you'll also be able to navigate to an Image Lab offering a number of demos—for example, a color depth demo that lets you display a particular image at a number of different depths (from 2 to 8) and compare the results.
- **Software.** Freeware, shareware, or commercial demo packages for a variety of platforms (Microsoft Windows, MS-DOS, OS/2, the Macintosh, and UNIX), as well as source code (for such popular packages as pbmplus and xv, and for libraries of JPEG, PNG, TIFF, and other format functions).
- **Internet.** We're committed to keeping this product up to date. If you have an Internet connection, you can link to the GFF Home Page to see what's new at the GFF Web Center. There you'll find updates to the product (new file formats, code, and images), additional information about computer graphics and file formats (e.g., Frequently Asked Questions listings [FAQs], graphics news), and links to updated versions of the specs and software included on our CD-ROM.

Ex.1007 / Page 19 of 1158
APPLE INC.

- **The Book.** The complete online text of Parts One and Three of the book, with cross-references and links that will help you navigate more easily through the text.
- **Index.** A complete index to the contents of the product.

We could say a lot more about what's available and how it all works, but the best way to find out is to jump in and try it for yourself.

Who Is the Book For?

This book is primarily for graphics programmers, but it's also for application programmers who need to become graphics programmers (if only for a little while). Although we didn't anticipate, in the first edition, that the book would be useful to graphics illustrators, we found that it was. In this second edition of the book and the CD-ROM, we've tried to provide additional resources for this audience. The book is also for anyone who needs a quick way to identify a graphics file of unknown origin. If you're not a graphics programmer, but want to get up to speed quickly, you'll find that Part One of the book requires little prior knowledge of computer graphics. It will help you become familiar with concepts associated with the storage of graphics data. In fact, a working knowledge of a programming language is useful, but not absolutely essential, if you're only looking for the big picture.

If you just want some background on graphics file formats, you might want to read Part One and refer, as needed, to the articles in Part Two and the appendices in Part Three. If you're in search of implementation guidance, you will want to refer to the articles and example code. Of course if you're a computer graphics professional, you might be interested primarily in the specification documents and tools on the CD-ROM accompanying this book.

In the unlikely event that you are creating your own new graphics file format, we fervently hope that this book provides you with some perspective on your task, if only by exhibiting the decisions—good and bad—that are frozen in the formats described in these pages.

How to Use the Book

This book is divided into three parts.

Part One, Overview, is an introduction to those computer graphics concepts that are especially helpful when you need to work with graphics file formats.

- Chapter 1, *Introduction,* introduces some basic terminology, and gives an overview of computer graphics data and the different types of graphics file

Ex.1007 / Page 20 of 1158
APPLE INC.

formats used in computer graphics. This chapter also lists all of the formats described in this book.

- Chapter 2, *Computer Graphics Basics*, discusses some concepts from the broader field of computer graphics that are necessary for an understanding of the rest of the book.
- Chapter 3, *Bitmap Files*, describes the structure and characteristics of bitmap files.
- Chapter 4, *Vector Files*, describes the structure and characteristics of vector files.
- Chapter 5, *Metafiles*, describes the structure and characteristics of metafiles.
- Chapter 6, *Platform Dependencies*, describes the few machine and operating system dependencies you will need to understand.
- Chapter 7, *Format Conversion*, discusses issues to consider when you are converting between the different format types (e.g., bitmap to vector), as well as between formats within a type (e.g., vector to vector).
- Chapter 8, *Working With Graphics Files*, describes the issues that come up when you read, write, and test graphics files. It also covers the corruption and encryption of graphics files, the potential for virus infection in those files, and the issues involved in writing your own file formats and file format specifications, including copyright issues.
- Chapter 9, *Data Compression*, describes data compression, particularly as compression techniques apply to graphics data and the graphics files described in this book.
- Chapter 10, *Multimedia*, surveys multimedia formats and issues.

Part Two, Graphics File Formats, describes the graphics file formats themselves. There is one article per format or format set, and articles are arranged alphabetically. Each article provides basic classification information, an overview, and details of the format. In many cases we've included short code examples. We've also indicated whether the specification itself (or an article that describes the details of the format) is included on the CD-ROM that accompanies this book, as well as code examples and images encoded in that format. Also provided in the articles are references for further information.

Part Three, Appendices, contains the following material:

- Appendix A, *Graphics Files and Resources on the Internet*, describes how to use a variety of information services on the Internet (email, USENET, FTP, Archie, and the World Wide Web) to obtain, post, and otherwise deal with

Ex.1007 / Page 21 of 1158
APPLE INC.

graphics files. It includes a listing of recommended sources of information about computer graphics and graphics file formats.

- Appendix B, *Graphics Files and Resources on the Commercial Services*, provides pointers to information about graphics files and resources on CompuServe, America Online, and a variety of bulletin board systems (BBSs).
- Appendix C, *Installation and Setup*, describes how to get the online product up and running on your system.

We also include a Glossary, which gives definitions for terms in the text.

"For Further Information" sections throughout the book list suggestions for further reading.

Conventions Used in This Book

We use the following formatting conventions in this book:

- **Bold** is used for headings in the text.
- *Italics* are used for emphasis and to signify the first use of a term. Italics are also used for functions, email addresses, FTP sites, directory and filenames, and newsgroups.
- All code and header examples are in `Constant Width`.
- All numbers in file excerpts and examples are in hexadecimal unless otherwise noted.
- All code and header examples use the following portable data types:

`BYTE`	8-bit unsigned data
`CHAR`	8-bit signed data
`WORD`	16-bit unsigned integer
`SHORT`	16-bit signed integer
`DWORD`	32-bit unsigned integer
`LONG`	32-bit signed integer
`FLOAT`	32-bit single-precision floating point number
`DOUBLE`	64-bit double-precision floating point number

All source code that we have produced is written in ANSI C. (This is relevant only if you are still using one of the older compilers.)

Ex.1007 / Page 22 of 1158
APPLE INC.

- All World Wide Web and FTP sites are listed in URL format, as shown below:

  protocol://site.name/
  protocol://site.name/directory/
  protocol://site.name/directory/file.name

Terminology of Computer Graphics

Computer graphics is in flux, and people working in the field are still busy creating vocabulary by minting new words. But they're also mutating the meanings of older words—words that once had a clear definition and context. Computer graphics is also an emerging field, in the sense that it is one fertilized by electronics, photography, film, animation, broadcast video, sculpture, and the traditional graphic arts. Each one of these fields has its own terminology and conventions, which computer graphics has inherited to some degree.

Complicating matters is that we're now in the era of electronic graphic arts. Color display adapters and frame buffers, paint and imaging programs, scanners, printers, video cameras, and video recorders are all being used in conjunction with the computer for the production of both fine and commercial art. A glance at any glossy magazine ad should give you some idea about how pervasive the mixing of digital and traditional media has become, if only because the overwhelming majority of magazines are now digitally composed. Indeed, the distinctions between traditional and computer art are becoming blurred.

Today we can find graphic artists producing work in traditional media, which is scanned into digital form, altered, re-rendered with a computer, and then distributed as original. While this is not a problem in itself, it nonetheless accelerates the injection of traditional terminology into computer graphics, countering any trend toward standardization. This will inevitably cause contradictions. Some are already apparent, in fact, and you'll probably notice them when we discuss the details of the formats.

There is no single consistent set of terms used across all of computer graphics. It is customary to cite standard references (like the classic *Computer Graphics: Principles and Practice* by James D. Foley, Andries vanDam, et al.) when arguing about terminology, but this approach is not always appropriate. Our experience is that usage in this field both precedes and succeeds definition. It also proceeds largely apart from the dictates of academia. To make matters worse,

Ex.1007 / Page 23 of 1158
APPLE INC.

the sub-field of graphics file formats is littered with variant jargon and obsolete usage. Many of the problems programmers have implementing formats can be traced to terminological misunderstandings.

In light of this, we have chosen to use a self-consistent terminology that is occasionally at odds with that of other authors. Sometimes, we have picked a term because it has come into common use, displacing an older meaning. An example of this is *bitmap*, which is now often used as a synonym for *raster*, making obsolete the older distinction between *bitmap* and *pixelmap*. Occasionally, we have been forced to choose from among a number of terms for the same concept. Our decision to use the term *palette* is one example of this.

For some of the same reasons, we use the term *graphics*, and avoid *graphic* and *graphical*. We all have to face up to the fact that the field is known as *computer graphics*, establishing a persistent awkwardness. We have chosen to use graphics as a noun as well as an adjective.

We believe that the choices we made represent a simplification of the terminology, and that this shouldn't be a problem if you're already familiar with alternate usage. Should you have any questions in this area, our definitions are available in the Glossary.

About the File Format Specifications

In preparing this book, we have made a monumental effort to collect, all in one place, the myriad graphics file format specifications that have until now been floating on the Internet—hiding in the basements of various organizations, gathering dust on individual application authors' bookshelves and in their private directories. We've done our best to locate the specifications and their caretakers (perhaps the original author, and perhaps the vendor that now maintains or at least owns the specification) and to obtain permission to include these documents on the CD-ROM that accompanies this book. In most cases, we have been able to obtain permission, but in some cases we have not.

There were several reasons for our failure to gain permission, some simple and some more complex. Although neither of us is a lawyer (or a bureaucrat!) or particularly interested in legal issues, we did encounter some legalities while gathering these specifications. Given our special perspective on the world of graphics file formats, we want to share our reactions to these legalities with you—perhaps in the hope that we'll see fewer problems in the future.

Here are the reasons why we couldn't include certain format specifications on the CD-ROM; here, we use the word *caretaker* to indicate either the author or owner of the specification or the organization that now has responsibility for its maintenance or distribution.

Ex.1007 / Page 24 of 1158
APPLE INC.

- **We couldn't find the caretaker.** We simply couldn't find out who owned the specification of some of the formats we knew about. This may or may not have been the vendor's fault, but try as we did, we just couldn't find the information. Here's where you can help us. If you know of a format that you yourself find useful, let us know what it is and how you think we might be able to obtain permission to include it in a future edition of this book.
- **The caretaker couldn't find the specification.** Strange, but true. This happened twice. To be honest, these were both small companies. But still. . .
- **We couldn't get past caretaker bureaucracy.** In some cases, we simply couldn't get through to the correct person in the organization in many months of trying. We know it's hard to believe. It seems that you could walk into any installation and in a few minutes figure out who knows what and where they are. We thought so too before we started this project. In fact, executive management at several vendors professed a willingness to provide us with information, but simply couldn't figure out how to do so. Here too, maybe our readers can help. . .
- **The caretaker wouldn't allow us to include the format.** In some cases, we found this reasonable. One obvious case was the BRL-CAD specification, which is massive and readily available. The U.S. government will send it to you if you ask for it. Other companies prefer to license the information as part of a developer's kit. Still others wished to restrict the currency of older formats, presumably so they wouldn't be bothered by users calling them up about them. Although we are philosophically in disagreement with vendors in this latter group, we are willing to admit that they have a point. Some companies, however, feel that releasing information on their formats would somehow give their competitors an advantage or would otherwise be to their own disadvantage. We hope they'll change their minds when they see how many other formats are represented here and how useful this compendium is to everyone—programmers and vendors alike. Finally, several vendors have taken the most extreme position that information on their formats is proprietary and have used legal means to prevent developers from working with them. This last case is the most alarming, and we discuss it further below.

We find it hard to understand why vendors have patented their formats and/or used contract law arguments to restrict access to information on their formats. It seems obvious enough to us—and to others in the industry—that the way to get people to purchase your products is to make them easy to work with, not only for users, but for developers, too. Historically, this has been the case. Vendors who locked up their systems necessarily settled for a smaller share of the market.

Ex.1007 / Page 25 of 1158
APPLE INC.

Although some vendors seem nearly paranoid, we suspect that the majority that restrict their formats don't have a clear idea what they're selling. This is particularly true for vendors of large, vertically integrated systems, where the format plays a small, but key, role in the overall product strategy.

Nevertheless, whether justified in our view or not, the restriction is real and serves as an alarming and dangerous precedent. As the various parts of the computer industry converge, competition for market share is necessarily increasing. There is a tendency for entities in the business to grow larger and more corporate. What one company does, its competitors must do to stay in the market. At least they consider doing it.

Now, the reality of the situation is that no vendor can restrict information totally and indefinitely. This is particularly the case with file formats. Vendors generally seek to protect their formats through a combination of encryption and legal remedies. However, a person who buys the application which produces the restricted format as output buys a generator of an infinite number of samples. Because applications are judged, among other things, by the speed with which they produce output, and because encryption and obfuscation schemes take time to both implement and use, not much time and effort has gone into making formats unbeatable. To date, encrypted and obfuscated formats have been pretty easy to crack.

An example that comes to mind is Adobe's Type 1 font format encryption. This was used by Adobe to protect its font outlines, but knowledge of the encryption scheme was fairly widespread in certain commercial circles before Adobe publicized it. Whether this resulted in commercial losses to Adobe from piracy of their outlines is hard to say. It certainly generated a good deal of ill will in the industry and ultimately proved futile.

This being the case, some vendors have taken to the courts to protect their formats. We find this behavior futile and ill-conceived. Even if it has a short-term benefit on revenues, the long-term losses due to a restricted market and developer ill-will seem to outweigh this benefit. In a sense, it is a form of monopolistic behavior, or certainly a type of positioning designed to support future monopolistic behavior.

Now, it's a fact of life that almost every format that has made it to the market has been reverse-engineered. This has seldom been for profit—more for the challenge. If you truly have a need to track down the information, it's certain that it can be found through the Internet, provided the information exists.

Ex.1007 / Page 26 of 1158
APPLE INC.

Is it legal to possess this information? This isn't clear at this time. Certainly it's illegal if the information was stolen from a vendor prior to publication. We, by the way, know of no instance where a restricted format has ever been stolen from a vendor. If you use or publicize information a vendor has tried to restrict legally, however, you run the risk of becoming involved in the legal affairs of the vendor, regardless of how that information was obtained. We do wish to point out that the legal way to influence the behavior of a commercial entity is in the marketplace.

The best-known vendor in recent years that has tried to restrict developers, through legal means, from obtaining information on its format is Kodak in the case of Photo CD, as we describe in the Kodak Photo CD article in Part Two.

In summary, although we could not include information on several of the formats we might have wished to, that information is almost surely available somehow for you to study so you'll understand more about format construction. However, if the format is legally restricted, you probably can't use it in your application, and there's no use thinking otherwise.

About the Examples

You'll find short code examples associated with some of the articles, but in most cases the full examples are not included in the file format articles themselves. We have done this mainly because many of the code examples are quite long, and we wanted to make it easier to find information in the book. All of the code is included on the accompanying CD-ROM.

The examples are, in most cases, C functions which parse and read (or write) format files. The examples are just that—examples—and are meant to give you a jump-start reading and writing image files. These are generally not standalone applications. In most cases, we wrote this code ourselves during the writing of the book or as part of other projects we've worked on. In some cases, code was contributed by other programmers or by those who own the file format specifications described in this book. We've also referred you to the source code for certain software packages on the CD-ROM that handle specific types of file formats—for example, the libtiff software, which provides extensive code illustrating the handling of TIFF files. These packages provide more extensive and useful examples.

Our own examples are usually written in a platform-independent manner. There is a bias for integer word lengths of 32 bits or less, for the simple reason that the overwhelming majority of files written to date have been on machines with a 32-bit or smaller word size. All examples and listings in this book and on the CD-ROM are written in ANSI C.

Ex.1007 / Page 27 of 1158
APPLE INC.

The code is provided for illustrative purposes only. In some cases, we have spent considerable time constructing transparent examples, and it's not necessarily an easy job. So be forewarned: if you use our code, absolutely no attempt has been made to optimize it. That's your job!

Can you use our code freely? In most cases, yes. We and O'Reilly & Associates grant you a royalty-free right to reproduce and distribute any sample code in the text and companion disk written by the authors of this book, provided that you:

- Distribute the sample code only in conjunction with and as a part of your software product
- Do not use our names or the O'Reilly & Associates name to market your software product
- Include a copyright notice (both in your documentation and as a part of the sign-on message for your software product) in the form:

```
Portions Copyright (C) 1994, 1996 by James D. Murray and
   William vanRyper
```

Please also note the disclaimer statements on the copyright page of this book.

Note as well that it is your responsibility to obtain permission for the use of any source code included on the CD-ROM that is not written by the authors of this book.

About the Images

Along with the specification documents and code, we have collected sample images for many of the graphics file formats. You can use these sample images to test whether you are successfully reading or converting a particular file format.

As we mentioned in a previous section, the online product also provides an Image Lab offering graphics demos that will let you view, compare, and compress images of different types, sizes, and depths.

About the Contributed Software

We are not the first programmers who have discovered how cumbersome and troublesome graphics file formats can be. We have elected to organize the chaos by writing a book. Other programmers among us have responded by writing software that reads, converts, manipulates, or otherwise analyzes

Ex.1007 / Page 28 of 1158
APPLE INC.

graphics files. Many of them have kindly agreed to let us include their software on the CD-ROM that accompanies this book. The packages we have elected to include provide an excellent sampling of what is available in the world of publicly available software. They include software for Windows (3.1, 95, and NT), MS-DOS, OS/2, the Macintosh, and UNIX. There are also several excellent packages that is provided as source code. Although this code is most often compiled for UNIX, it can (with some work) be used on other platforms as well.

Although many of these packages are readily available on the Internet or via various PC bulletin board systems, their freeware or shareware nature should not in any way suggest that they lack value. These are excellent packages, and we are very grateful that we have been able to include them here. They should help you considerably in your dealings with graphics files.

Which Platforms?

The online product runs on a variety of platforms—Windows 95, Windows NT, Windows 3.1, the Macintosh, and UNIX.

Most of the graphics file formats we describe in Part Two of this book originated on a particular platform for use in a particular application—for example, MacPaint on the Macintosh. Despite their origins, most files can be converted readily to other platforms and can be used with other applications, as we describe later in this book. There are a few issues that you need to be aware of, though, having to do with the platform on which you are working or the platform on which a particular graphics file was developed. These issues are summarized in Chapter 6.

Request for Comments

As you might imagine, locating and compiling the information that went into this book was no easy task, and in some cases our way was blocked, as we've discussed earlier. We're sure that some of the information we searched for is out there—somewhere!

We'd like to continue improving future editions, and for this we'll need all the help we can get. In addition to correcting any errors and omissions, we'd particularly like to expand our coverage of some of the more obscure graphics file formats that we might not know about or were unable to collect in time for publication. Also, if we were wrong, we want to know about it.

Ex.1007 / Page 29 of 1158
APPLE INC.

If you're in a position to help us out, or if you have any comments or suggestions on how we can improve things in any way, we'd love to hear from you. Please write, email, or call:

O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol CA 95472
800-998-9938 (in the U.S. or Canada)
707-829-0515 (international/local)
707-829-0104 (FAX)
Internet: *bookquestions@ora.com*
UUCP: *uunet!ora!bookquestions*

Acknowledgments

Writing this book, and collecting the voluminous material that is included on the CD-ROM that accompanies it, was a huge effort made possible only by the extraordinary generosity and common sense of a great many people. When we set out to collect the vast set of file format specifications in common use today, we frankly expected to be able to persuade only a small fraction of the specification owners and vendors that contributing them freely to this effort would be a good idea for them. We were convinced that it was a good idea, of course, but given the practicalities of bureaucracy, competition, and the many demands on people's time, we were not optimistic about conveying that conviction to others in the graphics community.

It was not an easy effort, and sometimes we nagged, wheedled, and otherwise made nuisances of ourselves, but people came through for us in the most remarkable way. To all those who contributed specifications, images, and their expertise to this effort, thank you. We hope it pays off by increasing the general awareness of people in the market and by substantially reducing the number of support calls to those who maintain the file formats. We have tried to list all those who helped us, but there were so many over such a long period of time that we are bound to leave a few names off the list. Don't be shy about reminding us; we'll include your names next time. To all those listed and unlisted, please accept our thanks.

First Edition

Individuals who helped us obtain and understand specifications include the following: Keith Alexander, Chris Allis, Jim Anderson, Tony Apodaca, Jim

Ex.1007 / Page 30 of 1158
APPLE INC.

Atkinson, Ron Baalke, David Baggett, Dan Baker, Cindy Batz, Gavin Bell, Steve Belsky, Celia Booher, Kim Bovic, Neil Bowers, John Bridges, Richard Brownback, Rikk Carey, Steve Carlsen, Timothy Casey, Wesley Chalfant, Buckley Collum, Freda Cooke, Catherine Copetas, António Costa, Stephen Coy, John Cristy, William Darnall, Ray Davis, Tom Davis, Bob Deen, Michael Dillon, Shannon Donovan, John Edwards, Jerry Evans, Lee Fisher, Jim Fister, Chad Fogg, Michael Folk, Roger Fujii, Jean-loup Gailly, Gary Goelhoeft, Bob Gonsales, Debby Gordon, Hank Gracin, Joy Gregory, Scott Gross, Hadmut, Paul Haeberli, Eric Haines, Eric Hamilton, Kory Hamzeh, William Hanlon, Fred Hansen, Paul Harker, Chris Hecker, Bill Hibbard, Michael Hoffman, Steve Hollasch, Terry Ilardi, Neale Johnston, Mike Kaltschnee, Lou Katz, Jim Kent, Pam Kerwin, Craig Kolb, Steve Koren, Don Lancaster, Tom Lane, Ian Lepore, Greg Leslie, Glenn Lewis, Paul Mace, Britt Mackenzie, Mike Martin, Pat McGee, Brian Moran, Mike Muuse, JoAnn Nielson, Gail Ostrow, Joan Patterson, Jeff Parker, Brian Paul, Brad Pillow, Andrew Plotkin, Jef Poskanzer, John Rasure, Dave Ratcliffe, Jim Rose, Randi Rost, Carroll Rotkel, Stacie Saccomanno, Jim Saghir, Barry Schlesinger, Louis Shay, Bill Shotts, Rik Segal, Mark Skiba, Scott St. Clair, John Stackpole, Marc Stengel, Ann Sydeman, Mark Sylvester, Spencer Thomas, Mark VandeWettering, Greg Ward, Archie Warnock, David Wecker, Joel Welling, Drew Wells, Jocelyn Willett, James Winget, and Shiming Xu.

Thanks to these organizations that shared their specifications and helped us in other ways: 3D/Eye, Adobe Systems, Aldus, Andrew Consortium, Apple Computer, Autodesk, Avatar, Carnegie Mellon University, C-Cube Microsystems, Commodore Business Machines, CompuServe, Computer Associates, Digital Equipment Corporation, Digital Research, DISCUS, DuPont, IBM, Inset Systems, Intel Corporation, ISEP/INESC, Jet Propulsion Laboratory, Khoral Research, Kofax Image Products, Kubota Pacific Computer, Lawrence Berkeley Laboratory, Massachusetts Institute of Technology, Media Cybernetics, Metron Computerware, Microsoft, National Aeronautics and Space Administration, National Center for Supercomputer Applications, National Oceanic and Atmospheric Administration, Novell, Paul Mace Software, Pittsburgh Supercomputing Center, Pixar, Princeton University Department of Computer Science, RIX SoftWorks, Silicon Graphics, Sun Microsystems, Time Arts, Truevision, University of Illinois, University of Utah Department of Computer Science, University of Wisconsin at Madison, U.S. Army Ballistic Research Lab, U.S. Army Research Laboratory, Wavefront Technologies, X Consortium, and ZSoft.

Very special thanks to those who read the manuscript of this book and made thoughtful and helpful comments on the content and organization: Dr. Peter Bono, Jim Frost, Tom Gaskins, Lofton Henderson, Dr. Tom Lane, Tim

Ex.1007 / Page 31 of 1158
APPLE INC.

O'Reilly, Sam Leffler, Dr. Anne Mumford, and Archie Warnock. Double thanks to Lofton Henderson, who prepared CGM images for us under a tight deadline, and to Tom Lane, who reviewed the CD-ROM as well. We could not have done this without you!

Many thanks to Shari L.S. Worthington, who let us include in this book many of the Internet resources she compiled for her article, "Imaging on the Internet: Scientific/Industrial Resources" in *Advanced Imaging*, February 1994.

We are very grateful to those who contributed software that will allow you to convert, view, manipulate, and otherwise make sense of the many file formats described in this book. We could not include all of these packages on the CD-ROM (for this version, at least), but we appreciate very much your generosity. Thanks to these individuals: Dan Baker, Robert Becker, Alexey Bobkov, John Bradley, Mike Castle, John Cristy, Orlando Dare, D.J. Delorie, Guiseppe Desoli, Chris Drouin, Stefan Eckart, Mark Edmead, Chad Fogg, Oliver Fromme, Mike Fitzpatrick, Jim Frost, Aaron Giles, Graeme Gill, Maynard Handley, Jih-Shin Ho, Robert Holland, David Holliday, Allen Kempe, Andrew Key, Michail Kutzetsov, Tom Lane, Sam Leffler, Thorsten Lemke, Leonardo Loureiro, Eric Mandel, Michael Mauldin, Frank McKenney, Kevin Mitchell, Bob Montgomery, David Ottoson, Jef Poskanzer, Igor Plotnikov, Eric Praetzel, Wayne Rasband, Mohammed Ali Rezaei, Rich Siegel, Davide Rossi, Y. Shan, Doug Tody, Robert Voit, Archie Warnock, Ken Yee, Norman Yee, and Paul Yoshimune. And thanks to these organizations: Alchemy Mindworks, Bare Bones Software, Carnegie Mellon University, CenterLine Software, Express Compression Labs, DareWare, Goddard Space Center, Handmade Software, Honeywell Technology Center, Independent JPEG Group, Labtam Australia Pty Ltd, MTE Industries, National Institutes of Health, National Optical Astronomy Observatories, Peepworks, PixelVision Software, Phase II Electronics, Stoik Ltd, TBH-SoftWorx, University of Waterloo. Special thanks to Rich Siegel and Leonard Rosenthal, who offered the use of their BBEdit and Stuffit products, as well as their knowledge of the Macintosh.

Many, many thanks to all the good people at O'Reilly & Associates who made this book happen: to our editor, Debby Russell, who guided this effort from beginning to end; to Gigi Estabrook, who tirelessly collected permissions and documents from the many vendors; to Ellen Siever who as production manager got an enormous book out under incredible time pressure; to Nicole Gipson, Jessica Hekman, Mary Anne Weeks Mayo, Kismet McDonough, Clairemarie Fisher O'Leary, and Stephen Spainhour who worked on the production team; to Chris Tong, who produced the index; and to Chris Reilley, who created the figures. Special thanks to Len Muellner and Norm Walsh who spent many long hours developing tools, filters, and magical potions that tamed the SGML beast.

Ex.1007 / Page 32 of 1158
APPLE INC.

We are also very grateful to those who produced the CD-ROM that accompanies this book: to Debby Russell, who headed up the effort; to Terry Allen, who converted and otherwise beat the files into submission; to Norm Walsh and Jeff Robbins, who lent their expertise in PC and Macintosh environments; and to Dale Dougherty, Lar Kaufman, Linda Mui, Tim O'Reilly, Eric Pearce, and Ron Petrusha, who all contributed their knowledge of CD-ROM technology, their experiences with other projects, and their opinions about how we ought to proceed. Thanks to Jeff Moskow and Ready-to-Run Software, who did the final CD-ROM development for us.

A special thank you to P.J. Mead Books & Coffee of Orange, California, where James D. Murray sought caffeine and solace during the long days and nights of writing and revising the first edition of this book.

Second Edition

Many people helped us to prepare this second edition of the book. We are grateful to those who helped us to update information on the file formats included in the original edition: David Baggett, Greg Broiles, Antonio Costa, Mike Fitzpatrick, Eric Haines, Lofton Henderson, Dale Holt, Tom Lane, Glenn Lewis, David Spoelstra, Greg Ward, Joel Welling, and Ann Marie Zanger,

We are also grateful to those who helped us acquire new file format specifications and sample code and images: David Charlap, Jennifer Cohan, Andreas Dilger, Jon Erikson, Matt Foster, Greg Gilley, Bert Greevenbosch, Eric Hamilton, Jaromir Krejci, Tom Lane, Ken Lunde, Jonathan Nimer, Jolyon Ralph, Glenn Randers-Pehrson, Greg Roelofs, Hal Stern, and Ernie Wright,

Thanks to those who reviewed chapters and sections for us during the preparation of this book: David Charlap, John Foust, Eric Hamilton, Simson Garfinkel, Bert Greevenbosch, Tom Lane, Jamomir Krejci, Andreas Dilger, Glenn Randers-Pehrson, Greg Roelofs, Jolyon Ralph, Gene Spafford, and Ernie Wright,

John Foust also helped in the writing of several of the file format articles.

Thanks also to Nathan Cass and Mark Starr of Unisys who helped us to understand the complexities of the Unisys/CompuServe/LZW patent issue; to John Dockery who did online CompuServe and AOL research for us; and to Paula Ferguson and Andrew Schulman of ORA who advised us about certain formats.

Special thanks to those who reviewed the product and the CD-ROM at various stages: Lee Estrin, Kerrie Kennedy, and Mike Kaltschnee. And thanks to Ken DeCanio, who did the Macintosh software development for us.

Ex.1007 / Page 33 of 1158
APPLE INC.

We are grateful to the many software authors who allowed us to include their packages on the new version of the CD-ROM. Mark Adler, Daniel Baker, Sean Barger, John Bradley, Ken Carlino, Bill Cotton, John Cristy, Lee Daniel Crocker, Stefan Eckart, Debbie Eddy, Michael Fitzpatrick, R. Mark Fleming, Chad Fogg, Martin Fong, Oliver Fromme, Jim Frost, Jean-loup Gailly, Gary Gehman, Aaron Giles, Graeme Gill, Ed Hamrick, Maynard Handley, Paul Ho, Gareth Hunt, Peter Lerup, Chris Komnic, Markus Kuhn, Andreas Lampen, David Lane, Tom Lane, Sam Leffler, Alexander Lehmann, Thorsten Lemke, Andreas Ley, Leonardo Hadddad Loureiro, Michael Mauldin, Kelly McKiernan, Douglas Mink, Kevin Mitchell, John Montbriand, Bob Montgomery, Joe Oliphant, Yves Piguet, Igor Plotnikov, Jef Poskanzer, Wayne Rasband, Steve Rimmer, Keith Rule, Jonathan Schafer, Guy Eric Schalnat, Paul Schmidt, David Schooley, Shiva Shenoy, Eric Toonen, Willem A.J. van Schaik, Archibald Warnock, Kevin Woolley, Ken Yee, and Norman Yee.

And thanks to the organizations that helped us in various ways to acquire or review specs, supply software, and otherwise move this project along: 3D/Eye, Adobe, Advance Enterprises, Alchemy Mindworks, Anagrafyxx Software, Andrew Consortium, Artemis Software, AUI/NRAO, Beyond Midnight Software, Boxtop Software, Dr. Dobb's Journal, Equilibrium, Group 42, Hamrick Software, HyperAct, Inset, i2 Technologies, ISEP/INESC, JASC Inc., Kamyan Software, Lemke Software, Magic Bullet Communications, Mmedia Research Corporation, National Institutes of Health, NOAO/IRAF, Photodex, Piclab, Pittsburgh Supercomputing Center, Quarterdeck, Smithsonian Astrophysical Observatory, Spyglass, St. Mary's University, Sun Microsystems, Sverdlup Technology, TrueVision, Ulead Systems, Unisys, Woolleysoft, and Zoner.

And, once again, all our thanks to the people at O'Reilly & Associates who turned into reality the vision of making this book into a true online product: Linda Walsh, the product manager; Norm Walsh, who built the CD-ROM/Internet product; Edie Freedman, who designed the user interface and the overall look and feel of the product, as well as the book cover; Debby Russell, our editor once again; Gigi Estabrook, who wrote online text and organized the software; Sue Willing, who acquired permissions and did some writing as well; Sheryl Avruch, the keeper of the schedule and the director of production; David Futato, the production manager for the book; Nancy Priest, who designed the interior format; Chris Reilley, who did the figures; and Seth Maislin, who produced the index.

Ex.1007 / Page 34 of 1158
APPLE INC.

Finally,

This book is dedicated to my son, James Alexander Ozbirn Murray, and his mother, Katherine.

James D. Murray

To all sentient beings working with graphics file formats.

William vanRyper

Ex.1007 / Page 35 of 1158
APPLE INC.

PART ONE

Overview

CHAPTER 1

Introduction

A graphics file format is the format in which graphics data—data describing a graphics image—is stored in a file. Graphics file formats have come about from the need to store, organize, and retrieve graphics data in an efficient and logical way. Sounds like a pretty straightforward task, right? But there's a lot under the covers, and that's what we're going to talk about.

File formats can be complex. Of course they never seem complex until you're actually trying to implement one in software. They're also important, in ways that often aren't obvious. You'll find, for instance, that the way a block of data is stored is usually the single most important factor governing the speed with which it can be read, the space it takes up on disk, and the ease with which it can be accessed by an application. A program simply must save its data in a reasonable format. Otherwise, it runs the risk of being considered useless.

Practically every major application creates and stores some form of graphics data. Even the simplest character-mode text editors allow the creation of files containing line drawings made from ASCII characters or terminal escape sequences. Graphical user interface (GUI)–based applications, which have proliferated in recent years, now need to support hybrid formats to allow the incorporation of bitmap data in text documents. Database programs with image extensions also let you store text and bitmap data together in a single file. In addition, graphics files are an important transport mechanism that allows the interchange of visual data between software applications and computer systems.

There is currently a great deal of work being done on object-based file systems, where a "data file" may appear as a cluster of otherwise unrelated elements and may be just as likely to incorporate graphics data as not. Clearly, traditional

Ex.1007 / Page 37 of 1158
APPLE INC.

data classification schemes are in need of revision. Nevertheless, there will remain an enormous amount of graphics data accessible only by virtue of our ability to decode and manipulate the files we find around us today.

The Basics

Before we explore the details of any particular file formats, we first need to establish some basic background and terminology. Because we're assuming you have a general working knowledge of computers and know some programming, we'll start with some definitions. You'll find that we've simplified and condensed some of the terminology found in standard computer graphics references. The changes, however, always reflect modern usage. (You'll find a discussion of our rationale in the Preface.)

In what follows, we will be speaking of the *output* of a computer graphics process, or the production of a *graphic work* by a program. We don't mean to seem anthropomorphic here. The author of the work is usually human, of course, but his or her contribution is *input* from the point of view of this book. We're mainly concerned about the portion of the output that comes from a program and winds up in a file. Because the program is the last thing that "touches" the data before it winds up on disk or tape, we say that a graphic work is produced by a program, rather than by a human being. (In this case, we are referring to the form in which the data is stored, and not its meaning or content.)

Graphics and Computer Graphics

Traditionally, *graphics* refers to the production of a visual *representation* of a real or imaginary object created by methods known to graphic artists, such as writing, painting, imprinting, and etching. The final result of the traditional graphics production process eventually appears on a 2D surface, such as paper or canvas. *Computer graphics*, however, has expanded the meaning of *graphics* to include any data intended for display on an *output device*, such as a screen, printer, plotter, film recorder, or videotape.

Notice what's happened here. Graphics used to refer to the *actual output*, something you could see. Now it means something only *intended* for display, or something meant to be turned into output. This distinction may seem silly to experienced users, but we've watched artists new to computers struggle with this. Where is the graphic output from a paint program? Does it appear as you compose something on the screen? Where is the representation when you write your work to a file? Does it appear for the first time when another program displays it on a screen or paper?

Ex.1007 / Page 38 of 1158
APPLE INC.

In the practice of computer graphics, creation of a work is often separate from its representation. One way to put it is that a computer graphics process produces *virtual output* in memory, or persistent output in a file on permanent media, such as a disk or tape. In other words, even though a program has written a file full of *something*, output doesn't yet exist from a traditional point of view because nothing has been displayed anywhere. So we say that *graphics data* is the *virtual output* of a program, from which a representation of the work can be constructed, or can be reconstructed from the persistent graphics data saved to a file, possibly by the same program.

Rendering and Images

In the interest of clarity, most people make a distinction between creation and *rendering* (sometimes also called *realization*). Traditionally, an *image* is a visual representation of a real-world object, captured by an artist through the use of some sort of mechanical, electronic, or photographic process. In computer graphics, the meaning of an image has been broadened somewhat to refer to an object that appears on an output device. Graphics data is rendered when a program draws an image on an output device.

You will also occasionally hear people speak of the computer graphics *production pipeline*. This is the series of steps involved in defining and creating graphics data and rendering an image. On one end of the production pipeline is a human being; on the other end is an image on paper, screen, or another device. Figure 1-1 illustrates this process.

Graphics Files

For the purposes of this book, graphics files are files that store any type of persistent graphics data (as opposed to text, spreadsheet, or numerical data, for example), and that are intended for eventual rendering and display. The various ways in which these files are structured are called graphics file formats. We will examine several categories of graphics file formats later in this chapter.

People sometimes talk of rendering an image to a file, and this is a common and perfectly valid operation. For our purposes, when an image is rendered to a file, the contents of that file become persistent graphics data. Why? Simply because the data in the file now needs to be re-rendered as virtual graphics data before you can see what it looks like.

Although the image is once again turned into graphics data in the process of rendering it to the file, it is now once more merely data. In fact, the data can now be of a different type. This is what happens in file conversion operations,

Ex.1007 / Page 39 of 1158
APPLE INC.

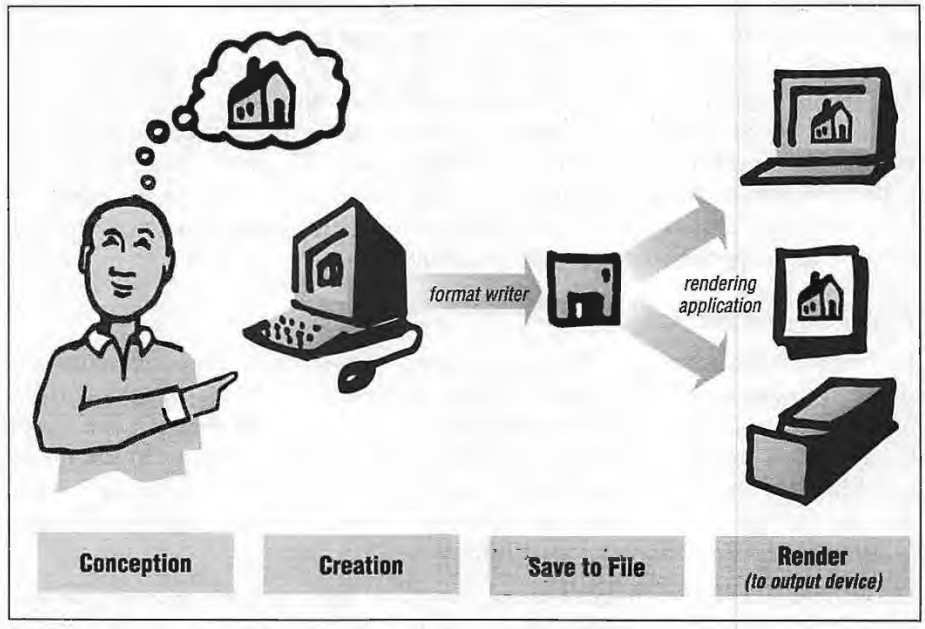


FIGURE 1-1: *The graphics production pipeline*

for example. An image stored in a file of format type 1 is rendered (by a conversion program) to a second file of format type 2.

Which Graphics Files Are Included . . .

Although we've tried in this book to stick to formats that contain graphics data, we've also tried to make sure that they are used for data interchange between programs. Now, you'd think that it's always perfectly clear whether a file contains graphics data or not. Unfortunately, this isn't always the case. Spreadsheet formats, for instance, are sometimes used to store graphics data. And what about data interchange? A format is either used to transfer data from one program to another or it isn't, right? Again, it's not so simple.

Some formats, such as TIFF, CGM, and GIF, were designed for interprogram data interchange. But what about other formats, such as PCX, which were designed in conjunction with a particular program? There is no easy answer, but these two criteria—graphics data and data interchange—will take you a long way, and we've tried as much as possible to follow them here.

A complete list of all of the formats described in this book is contained below in the section called "Format Summary".

Ex.1007 / Page 40 of 1158
APPLE INC.

...And Which Are Not

For the purposes of this book, we're excluding three types of files that contain graphics data but are outside the scope of what we are trying to accomplish here: *output device language* files, *page description language* files, and *FAX* files.

Output device language files contain hardware-dependent control codes that are designed to be interpreted by an output device and are usually used to produce hardcopy output. We exclude these, because they usually have a short lifetime as temporary files and with few exceptions are not archived or exchanged with other machines. Another reason is practical: many of the hundreds of types of printers and plotters built over the years use vendor-specific control information, which the market has traditionally ignored. By far the most common output device languages in use are Printer Control Language (PCL) and variants, used to control the Hewlett-Packard LaserJet series of laser printers and compatibles, and Hewlett-Packard Printer Graphics Language (HPGL), used to control plotters and other vector devices.

Page description languages (PDLs) are sophisticated systems for describing graphical output. We exclude page description languages from our discussion, because the market is dominated by PostScript and because the specification is voluminous and extremely well-documented in readily available publications. We do, however, provide an article describing Encapsulated PostScript format; that article briefly discusses EPS, EPSF, and EPSI formats.

FAX-format files are usually program-specific, created by an application designed to support one or more FAX modems. There are many such formats, and we do not cover them, because they generally are not used for file exchange. We do, however, include a brief article on FAX formats that discusses some of the issues you'll face if you use these formats. The main problem encountered by people working with FAX formats is finding information about compression algorithms. We've included a chapter covering some of the more common compression formats you will encounter.

Graphics Data

Graphics data is traditionally divided into two classes: *vector* and *bitmap*. As we explain below, we use the term *bitmap* to replace the older term *raster*.

Vector Data

In computer graphics, *vector data* usually refers to a means of representing lines, polygons, or curves (or any object that can be easily drawn with lines) by

Ex.1007 / Page 41 of 1158
APPLE INC.

numerically specifying *key points*. The job of a program rendering this key-point data is to regenerate the lines by somehow connecting the key points or by drawing using the key points for guidance. Always associated with vector data is attribute information (such as color and line thickness information) and a set of conventions (or rules) allowing a program to draw the desired objects. These conventions can be either implicit or explicit, and, although designed to accomplish the same goals, are generally different from program to program.

Figure 1-2 shows several examples of vector data.

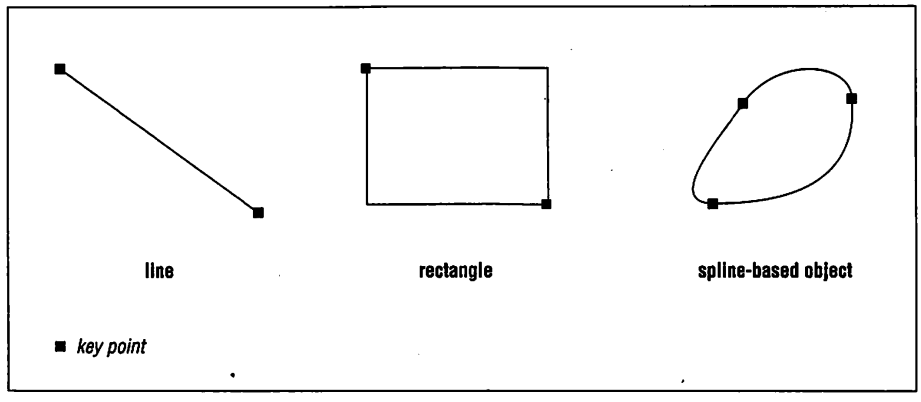


FIGURE 1-2: *Vector data*

By the way, you may be familiar with a definition of the word *vector* which is quite precise. In the sciences and mathematics, for instance, a vector is a straight line having both magnitude and direction. In computer graphics, *vector* is a sort of catch-all term. It can be almost any kind of line or line segment, and it is usually specified by sets of endpoints, except in the case of curved lines and more complicated geometric figures, which require other key points to be fully specified.

Bitmap Data

Bitmap data is formed from a set of numerical values specifying the colors of individual *pixels* or *picture elements* (*pels*). Pixels are dots of color arranged on a regular grid in a pattern representing the form to be displayed. We commonly say that a bitmap is an *array of pixels*, although a bitmap, technically, consists of an *array of numerical values* used to set, color, or "turn on" the corresponding pixels on an output device when the bitmap is rendered. If there is any ambiguity in the text, we will make the distinction clear by using the term *pixel value* to refer to a numerical value in the bitmap data corresponding to a pixel color in the image on the display device.

Ex.1007 / Page 42 of 1158
APPLE INC.

Figure 1-3 shows an example of bitmap data.

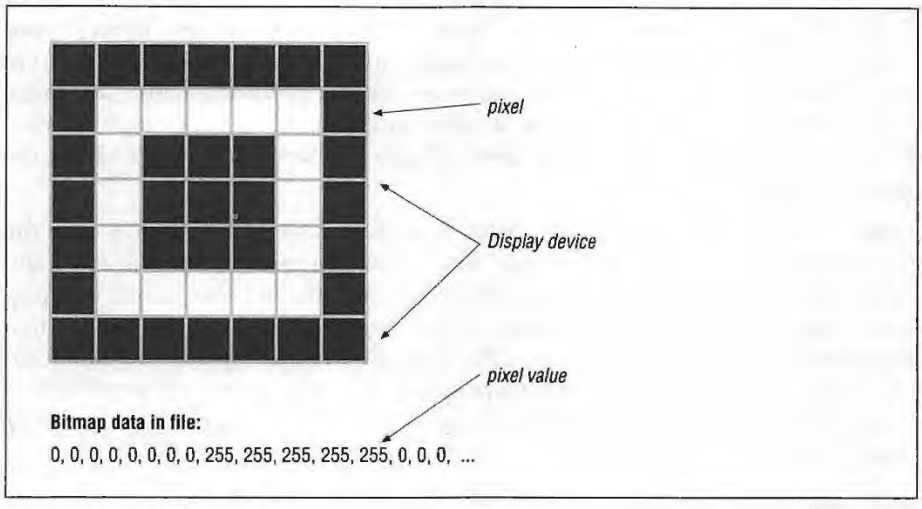


FIGURE 1-3: *Bitmap data*

In older usage, the term *bitmap* sometimes referred to an array (or "map") of single bits, each bit corresponding to a pixel, while the terms *pixelmap*, *graymap*, and *pixmap* were reserved for arrays of multibit pixels. We always use the term *bitmap* to refer to an array of pixels, whatever the type, and specify the *bit depth*, or *pixel depth*, which is the size of the pixels in bits or some other convenient unit (such as bytes). The bit depth determines the number of colors a pixel value can represent. A 1-bit pixel can be one of two colors, a 4-bit pixel one of 16 colors, and so on. The most commonly found pixel depths today are 1, 2, 4, 8, 15, 16, 24, and 32 bits. Some of the reasons for this, and other color-related topics, are discussed in Chapter 2, *Computer Graphics Basics.*

Sources of Bitmap Data: Raster Devices

Historically, the term *raster* has been associated with cathode ray tube (CRT) technology and has referred to the pattern of rows the device makes when displaying an image on a picture tube. Raster-format images are therefore a collection of pixels organized into a series of rows, which are called *scan lines.* Because raster output devices, by far the most popular kind available today, display images as patterns of pixels, pixel values in a bitmap are usually arranged so as to make them easy to display on certain common raster devices. For these reasons, bitmap data is often called *raster data.* We use the term *bitmap data* in this book.

Ex.1007 / Page 43 of 1158
APPLE INC.

As mentioned above, bitmap data can be produced when a program renders graphics data and writes the corresponding output image to a file instead of displaying it on an output device. This is one of the reasons bitmaps and bitmap data are often referred to as *images*, and bitmap data is referred to as *image data*. Although there is nothing to see in the traditional sense, an image can be readily reconstructed from the file and displayed on an output device. We will occasionally refer to the block of pixel values in a bitmap file as the *image* or *image portion*.

Other sources of bitmap data are raster devices used to work with images in the traditional sense of the word; raster devices are scanners, video cameras, and other digitizing devices. For our purposes, we consider a raster device that produces digital data to be just another source of graphics data, and we say that the graphics data is rendered when the program used to capture the data from the device writes it to a file. When speaking of graphics data captured from a real-world source, such as a scanner, people speak redundantly of a *bitmap image*, or an *image bitmap*.

What About Object Data?

People sometimes speak of a third class: *object data*. In the past, this referred to a method of designating complex forms, such as nested polygons, through a shorthand method of notation, and relying on a program's ability to directly render these forms with a minimal set of clues. Increasingly, however, the term is used to refer to data stored along with the program code or algorithmic information needed to render it. This distinction may become useful in the future, particularly if languages that support object-oriented programming (such as Smalltalk and C++) become more popular. However, for now we choose to ignore this third primitive data type, mainly because at the time of this writing, there are no standardized object file formats of any apparent commercial importance. In any case, the data portions of all objects can be decomposed into simpler forms composed of elements from one of the two primitive classes.

Figure 1-4 shows an example of object data.

Other Data

Graphics files may also include data containing structural, color, and other descriptive information. This information is included primarily as an aid to the rendering application in reconstructing and displaying an image.

Ex.1007 / Page 44 of 1158
APPLE INC.

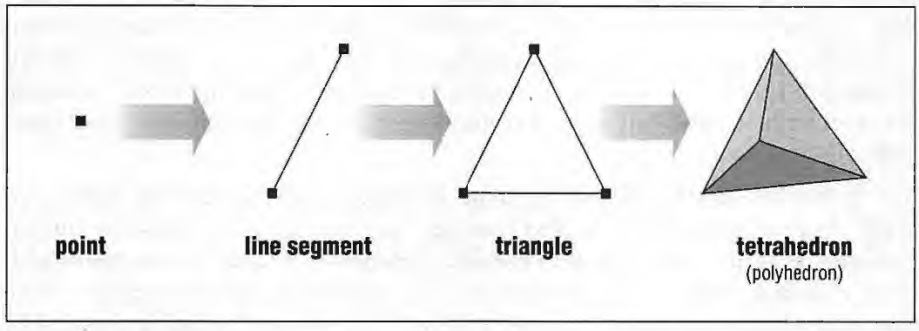


FIGURE 1-4: *Object data*

From Vector to Bitmap Data . . .

Twenty-five years ago, computer graphics was based almost entirely on vector data. Random-scan vector displays and pen plotters were the only readily obtainable output devices. The advent of cheap, high-capacity magnetic media, in the form of tapes and disks, soon allowed the storage of large files, which in turn created a need for the first standardized graphics file formats.

Today, most graphics storage is bitmap-based, and displays are raster-based. This is due in part to the availability of high-speed CPUs, inexpensive memory and mass storage, and high-resolution input and output hardware. Bitmap graphics are also driven by the need to manipulate images obtained from raster digitizing devices. Bitmap graphics are important in applications supporting CAD and 3D rendering, business charts, 2D and 3D modeling, computer art and animation, graphical user interfaces, video games, electronic document image processing (EDIP), and image processing and analysis.

It is interesting to note that the increased emphasis on bitmap graphics in our time corresponds to a shift toward the output end of the graphics production pipeline. By volume, the greatest amount of data being stored and exchanged consists of finished images in bitmap format.

The explosive growth of the World Wide Web has fueled this shift. Almost every Web page has one or more bitmap files associated with it. Bitmap images have become a part of everyday life for millions of people.

. . . And Back Again

But the trend toward bitmap data may not last. Although there are certain advantages to storing graphics images as bitmap data (we cover these in the

Ex.1007 / Page 45 of 1158
APPLE INC.

section called "Pros and Cons of Bitmap File Formats" in Chapter 3, *Bitmap Files*), bitmap images are usually pretty bulky. There is a definite trend toward networking in all the computer markets, and big bitmap files and low-cost networks don't mix. The cost of sending files around on the Internet, for example, can be measured not only in connect costs, but in lost time and decreased network performance.

The growth of the World Wide Web has accelerated this trend. The Web is currently built around HTML, a text markup language allowing software run on the machines of remote users to construct elaborate images of text pages with only minimal clues. This strategy, one of offloading imaging tasks to the machines of remote users to conserve network bandwidth, is being pursued by a number of vendors. This work is typified by the development of Java, Sun Microsystem's Internet programming language.

Because graphics files are surely not going to go away, we expect some sort of vector-based file format to emerge somewhere along the line as an interchange standard. Unfortunately, none of the present vector formats in common use is acceptable to a broad range of users (but this may change).

Types of Graphics File Formats

There are a number of different types of graphics file formats. Each type stores graphics data in a different way. Bitmap, vector, and metafile formats are by far the most commonly used formats, and we focus on these in this book. However, there are other types of formats as well—scene, animation, multimedia, hybrid, hypertext, hypermedia, 3D, virtual modeling reality language (VRML), audio, font, and page description language (PDL). The increasing popularity of the World Wide Web has made some of these formats more popular, and we anticipate increased interest in them in the future. Although most of these file types are outside the scope of this book, we do introduce them in this section.

Bitmap Formats

Bitmap formats are used to store bitmap data. Files of this type are particularly well-suited for the storage of real-world images such as photographs and video images. Bitmap files, sometimes called *raster files*, essentially contain an exact pixel-by-pixel map of an image. A rendering application can subsequently reconstruct this image on the display surface of an output device.

Microsoft BMP, PCX, TIFF, and TGA are examples of commonly used bitmap formats. Chapter 3 describes the construction of bitmap files in some detail.

Ex.1007 / Page 46 of 1158
APPLE INC.

Vector Formats

Vector format files are particularly useful for storing line-based elements, such as lines and polygons, or those that can be decomposed into simple geometric objects, such as text. Vector files contain mathematical descriptions of image elements, rather than pixel values. A rendering application uses these mathematical descriptions of graphical shapes (e.g., lines, curves, and splines) to construct a final image.

In general, vector files are structurally simpler than most bitmap files and are typically organized as data streams.

AutoCAD DXF and Microsoft SYLK are examples of commonly used vector formats. Chapter 4, *Vector Files*, describes the construction of vector files in some detail.

Metafile Formats

Metafiles can contain both bitmap and vector data in a single file. The simplest metafiles resemble vector format files; they provide a language or grammar that may be used to define vector data elements, but they may also store a bitmap representation of an image. Metafiles are frequently used to transport bitmap or vector data between hardware platforms, or to move image data between software platforms.

WPG, Macintosh PICT, and CGM are examples of commonly used metafile formats. Chapter 5, *Metafiles*, describes the construction of metafiles in some detail.

Scene Formats

Scene format files (sometimes called *scene description* files) are designed to store a condensed representation of an image or scene, which is used by a program to reconstruct the actual image. What's the difference between a vector format file and a scene format file? Just that vector files contain descriptions of portions of the image, and scene files contain instructions that the rendering program uses to construct the image. In practice it's sometimes hard to decide whether a particular format is scene or vector; it's more a matter of degree than anything absolute.

Animation Formats

Animation formats have been around for some time. The basic idea is that of the flip-books you played with as a kid; with those books, you rapidly displayed

Ex.1007 / Page 47 of 1158
APPLE INC.

one image superimposed over another to make it appear as if the objects in the image are moving. Very primitive animation formats store entire images that are displayed in sequence, usually in a loop. Slightly more advanced formats store only a single image but multiple color maps for the image. By loading in a new color map, the colors in the image change, and the objects appear to move. Advanced animation formats store only the differences between two adjacent images (called *frames*) and update only the pixels that have actually changed as each frame is displayed. A display rate of 10–15 frames per second is typical for cartoon-like animations. Video animations usually require a display rate of 20 frames per second or better to produce a smoother motion.

TDDD and TTDDD are examples of animation formats.

Multimedia Formats

Multimedia formats are relatively new but are becoming more and more important. They are designed to allow the storage of data of different types in the same file. Multimedia formats usually allow the inclusion of graphics, audio, and video information. Microsoft's RIFF, Apple's QuickTime, MPEG, and Autodesk's FLI are well-known examples, and others are likely to emerge in the near future. Chapter 10, *Multimedia,* describes various issues concerning multimedia formats.

Hybrid Formats

Currently, there is a good deal of research being conducted on the integration of unstructured text and bitmap data ("hybrid text") and the integration of record-based information and bitmap data ("hybrid database"). As this work bears fruit, we expect that hybrid formats capable of efficiently storing graphics data will emerge and will steadily become more important.

Hypertext and Hypermedia Formats

Hypertext is a strategy for allowing nonlinear access to information. In contrast, most books are linear, having a beginning, an end, and a definite pattern of progression through the text. Hypertext, however, enables documents to be constructed with one or more beginnings, with one, none, or multiple ends, and with many hypertext links that allow users to jump to any available place in the document they wish to go.

Hypertext languages are not graphics file formats, like the GIF or DXF formats. Instead, they are programming languages, like PostScript or C. As such, they are specifically designed for serial data stream transmission. That is, you can

Ex.1007 / Page 48 of 1158
APPLE INC.

start decoding a stream of hypertext information as you receive the data. You need not wait for the entire hypertext document to be downloaded before viewing it.

The term *hypermedia* refers to the marriage of hypertext and multimedia. Modern hypertext languages and network protocols support a wide variety of media, including text and fonts, still and animated graphics, audio, video, and 3D data. Hypertext allows the creation of a structure that enables multimedia data to be organized, displayed, and interactively navigated through by a computer user.

Hypertext and hypermedia systems, such as the World Wide Web, contain millions of information resources stored in the form of GIF, JPEG, PostScript, MPEG, and AVI files. Many other formats are used as well.

3D Formats

Three-dimensional data files store descriptions of the shape and color of 3D models of imaginary and real-world objects. 3D models are typically constructed of polygons and smooth surfaces, combined with descriptions of related elements, such as color, texture, reflections, and so on, that a rendering application can use to reconstruct the object. Models are placed in scenes with lights and cameras, so objects in 3D files are often called *scene elements.*

Rendering applications that can use 3D data are generally modeling and animation programs, such as NewTek's Lightwave and Autodesk's 3D Studio. They provide the ability to adjust the appearance of the rendered image through changes and additions to the lighting, textures applied to scene elements, and the relative positions of scene elements. In addition, they allow the user to animate, or assign motions to, scene elements. The application then creates a series of bitmap files, or frames, that taken in sequence can be assembled into a movie.

It's important to understand that vector data historically has been 2D in nature. That is, the creator application with which the data originated made no attempt to simulate 3D display through the application of perspective. Examples of vector data include CAD drawings and most clip art designed to be used in desktop publishing applications. There is a certain amount of confusion in the market about what constitutes 3D rendering. This is complicated by the fact that 3D data is now supported by a number of formats that previously stored only 2D vector data. An example of this is Autodesk's DXF format. Formats like DXF are sometimes referred to as *extended vector formats.*

Ex.1007 / Page 49 of 1158
APPLE INC.

Virtual Reality Modeling Language (VRML) Formats

VRML (pronounced "vermel") may be thought of as a hybrid of 3D graphics and HTML. VRML v1.0 is essentially a subset of the Silicon Graphics Inventor file format and adds to it support for linking to Uniform Resource Locators URLs in the World Wide Web.

VRML encodes 3D data in a format suitable for exchange across the Internet using the Hypertext Transfer Protocol (HTTP). VRML data received from a Web server is displayed on a Web browser that supports VRML language interpretation. We expect that VRML-based 3D graphics will soon be very common on the World Wide Web.

This book does not contain an in-depth discussion of VRML for some of the same reasons that we do not provide detailed descriptions of hypertext, hypermedia, and 3D formats. The VRML specification is a moving target, but you can keep up with it by looking at the following resources on the Internet:

http://www.oki.com/vrml/VRML_FAQ.html
VRML FAQ

http://www.sdsc.edu/vrml/
VRML information repositories

Audio Formats

Audio is typically stored on magnetic tape as analog data. For audio data to be stored on media such as a CD-ROM or hard disk, it must first be encoded using a digital sampling process similar to that used to store digital video data. Once encoded, the audio data can then be written to disk as a raw digital audio data stream, or, more commonly, stored using an audio file format.

Audio file formats are identical in concept to graphics file formats, except that the data they store is rendered for your ears and not for your eyes. Most formats contain a simple header that describes the audio data they contain. Information commonly stored in audio file format headers includes samples per second, number of channels, and number of bits per sample. This information roughly corresponds to the number of samples per pixel, number of color planes, and number of bits per sample information commonly found in graphics file headers.

Where audio file formats differ greatly is in the methods of data compression they use. Huffman encoding is commonly used for both 8-bit graphical and audio data. 16-bit audio data, however, requires algorithms specially adapted to

Ex.1007 / Page 50 of 1158
APPLE INC.

the problems of compressing audio data. Such compression schemes include the CCITT (International Telegraph and Telephone Consultative Committee) recommendations G.711 (uLAW), G.721 (ADPCM 32) and G.723 (ADPCM 24), and the U.S. federal standards FIPS-1016 (CELP) and FIPS-1015 (LPC-10E).

Because audio data is very different from graphics data, this book does not attempt to cover audio file formats. If you need more information on audio file formats, we recommend that you check out the following information resources on the Internet:

http://cuiwww.unige.ch/OSG/AudioFormats/
Guide to audio file formats

ftp://rtfm.mit.edu/pub/usenet/news.answers/audio-faq/part[1–2]
Audio file formats FAQ

ftp://rtfm.mit.edu/pub/usenet/news.answers/compression-faq/part[1–3]
comp.compression FAQ

ftp://rtfm.mit.edu/pub/usenet/news.answers/dsp-faq/part[1–4]
comp.dsp FAQ

ftp://rtfm.mit.edu/pub/usenet/news.answers/mpeg-faq/part[1–6]
MPEG FAQ

Font Formats

Another class of formats not covered in this book are font files. Font files contain the descriptions of sets of alphanumeric characters and symbols in a compact, easy-to-access format. They are generally designed to facilitate random access of the data associated with individual characters. In this sense, they are databases of character or symbol information, and for this reason font files are sometimes used to store graphics data that is not alphanumeric or symbolic in nature. Font files may or may not have a global header, and some files support sub-headers for each character. In any case, it is necessary to know the start of the actual character data, the size of each character's data, and the order in which the characters are stored in order to retrieve individual characters without having to read and analyze the entire file. Character data in the file may be indexed alphanumerically, by ASCII code, or by some other scheme. Some font files support arbitrary additions and editing, and thus have an index somewhere in the file to help you find the character data.

Some font files support compression, and many support encryption of the character data. The creation of character sets by hand has always been a

Ex.1007 / Page 51 of 1158
APPLE INC.

difficult and time-consuming process, and typically a font designer spent a year or more on a single character set. Consequently, companies that market fonts (called foundries for reasons dating back to the origins of printing using mechanical type) often seek to protect their investments through legal means or through encryption. In the United States, for instance, the names of fonts are considered proprietary, but the outlines described by the character data are not. It is not uncommon to see pirated data embedded in font files under names different from the original.

Historically there have been three main types of font files: bitmap, stroke, and spline-based outlines, described in the following sections.

We choose not to cover font files in this book because font technology is a world to itself, with different terminology and concerns. Many of the font file formats are still proprietary and encrypted and, in fact, are not available to the general public. Although there are a few older spline-based font formats still in use, font data in the TrueType and Adobe Type 1 formats is readily available on all the major platforms and is well-documented elsewhere in publications readily available to developers. We recommend that you check out the following resources on the Internet:

ftp://rtfm.mit.edu/pub/usenet/news.answers/fonts-faq/part[1–17]
Fonts FAQ

http://www.adobe.com
Fonts information repositories

Bitmap fonts

Bitmap fonts consist of a series of character images rendered to small rectangular bitmaps and stored sequentially in a single file. The file may or may not have a header. Most bitmap font files are monochrome, and most store fonts in uniformly sized rectangles to facilitate speed of access. Characters stored in bitmap format may be quite elaborate, but the size of the file increases, and, consequently, speed and ease of use decline with increasingly complex images.

The advantages of bitmap files are speed of access and ease of use—reading and displaying a character from a bitmap file usually involve little more than reading the rectangle containing the data into memory and displaying it on the display surface of the output device. Sometimes, however, the data is analyzed and used as a template for display of the character by the rendering application. The chief disadvantages of bitmap fonts are that they are not easily scaled, and that rotated bitmap fonts look good only on screens with square pixels.

Ex.1007 / Page 52 of 1158
APPLE INC.

Most character-based systems, such as MS-DOS, character-mode UNIX, and character terminal-based systems use bitmap fonts stored in ROM or on disk. However, bitmap fonts are seldom used today when sufficient processing power is available to enable the use of other types of font data.

Stroke fonts

Stroke fonts are databases of characters stored in vector form. Characters can consist of single strokes or may be hollow outlines. Stroke character data usually consists of a list of line endpoints meant to be drawn sequentially, reflecting the origin of many stroke fonts in applications supporting pen plotters. Some stroke fonts may be more elaborate, however, and may include instructions for arcs and other curves. Perhaps the best-known and most widely used stroke fonts were the Hershey character sets, which are still available online.

The advantages of stroke fonts are that they can be scaled and rotated easily, and that they are composed of primitives, such as lines and arcs, which are well-supported by most GUI operating environments and rendering applications. The main disadvantage of stroke fonts is that they generally have a mechanical look at variance with what we've come to expect from reading high-quality printed text all our lives.

Stroke fonts are seldom used today. Most pen plotters support them, however. You also may need to know more about them if you have a specialized industrial application using a vector display or something similar.

Spline-based outline fonts

Character descriptions in spline-based fonts are composed of control points allowing the reconstruction of geometric primitives known as splines. There are a number of types of splines, but they all enable the drawing of the subtle, eye-pleasing curves we've come to associate with high-quality characters that make up printed text. The actual outline data is usually accompanied by information used in the reconstruction of the characters, which can include information about kerning, and information useful when scaling characters that are very large or very small ("hints").

The advantages of spline-based fonts are that they can be used to create high-quality character representations, in some cases indistinguishable from text made with metal type. Most traditional fonts, in fact, have been converted to spline-based outlines. In addition, characters can be scaled, rotated, and otherwise manipulated in ways only dreamed about even a generation ago.

Ex.1007 / Page 53 of 1158
APPLE INC.

Unfortunatly, the reconstruction of characters from spline outline data is no trivial task, and the higher quality afforded by spline outlines comes at a price in rendering time and program development costs.

Page Description Language (PDL) Formats

Page description languages (PDLs) are actual computer languages used for describing the layout, font information, and graphics of printed and displayed pages. PDLs are used as the interpreted languages used to communicate information to printing devices, such as hardcopy printers, or to display devices, such as graphical user interface (GUI) displays. The greatest difference is that PDL code is very device-dependent. A typical PostScript file contains detailed information on the output device, font metrics, color palettes, and so on. A PostScript file containing code for a 4-color, A4-sized document can only be printed or displayed on a device that can handle these metrics.

Markup languages, on the other hand, contain no information specific to the output device. Instead, they rely on the fact that the device that is rendering the markup language code can adapt to the formatting instructions that are sent to it. The rendering program chooses the fonts, colors, and method of displaying the graphical data. The markup language provides only the information and how it is structured.

Although PDL files can contain graphical information, we do not consider PDLs to be graphics file formats any more than we would consider a module of C code that contains an array of graphical information to be a graphics file format. PDLs are complete programming languages, requiring the use of sophisticated interpreters to read their data; they are quite different from the much simpler parsers used to read graphics file formats.

Elements of a Graphics File

As mentioned in the Preface, different file format specifications use different terminology. In fact, it's possible that not a single term has a common meaning across all of the file formats mentioned in this book. This is certainly true for terms referring to the way data is stored in a file—terms such as *field*, *tag*, *block*, and *packet*. In fact, a specification will sometimes provide a definition for one of these terms and then abandon it in favor of a more descriptive one, such as a chunk, sequence, or record.

For purposes of discussion in this book, we will consider a graphics file to be composed of a sequence of data and data structures, called *file elements* or *data elements*. These are divided into three categories: fields, tags, and streams.

Ex.1007 / Page 54 of 1158
APPLE INC.

Fields

A *field* is a data structure that is a fixed size in a graphics file. A fixed field has not only a fixed size but a fixed position within the file. The location of a field is communicated by specifying either an absolute offset from a landmark in a file, such as the file's beginning or end, or a relative offset from some other data. The size of a field either is stated in the format specification or can be inferred from other information.

Tags

A *tag* is a data structure that can vary in both size and position from file to file. The position of a tag, like that of a field, is specified by either an absolute offset from a known landmark in the file, or through a relative offset from another file element. Tags themselves may contain other tags or a collection of related fields.

Streams

Fields and tags are an aid to random access; they're designed to help a program quickly access a data item known in advance. Once a position in a file is known, a program can access the position directly without having to read intervening data. A file that organizes data as a *stream*, on the other hand, lacks the structure of one organized into fields and tags and must be read sequentially. For our purposes, we will consider a stream to be made up of *packets*, which can vary in size, are sub-elements of the stream, and are meaningful to the program reading the file. Although the beginning and end of the stream may be known and specified, the location of packets other than the first usually is not, at least prior to the time of reading.

Combinations of Data Elements

You can imagine, then, pure fixed field files, pure tag files, and pure stream files, made up entirely of data organized into fixed fields, tags, and streams, respectively. Only rarely, however, does a file contain data elements of a single type; in most cases it is a combination of two or more. The TIFF and TGA formats, for example, use both tags and fixed fields. GIF format files, on the other hand, use both fixed fields and streams.

Fixed-field data is usually faster and easier to read than tag or stream data. Files composed primarily of fixed-field data, however, are less flexible in situations in which data needs to be added to or deleted from an existing file. Formats containing fixed fields are seldom easily upgraded. Stream data generally

Ex.1007 / Page 55 of 1158
APPLE INC.

requires less memory to read and buffer than field or tag data. Files composed primarily of stream data, however, cannot be accessed randomly, and thus cannot be used to find or sub-sample data quickly. These considerations are discussed further in Chapters 3, 4, and 5.

Converting Formats

You often need to convert a graphics file from one format to another—for printing, manipulation by a particular desktop publishing program, or some other reason. Although conversion to and from certain file formats is straightforward, conversion of other formats may be quite hair-raising. You will find conversion particularly problematic if you need to convert between basic format types—for example, bitmap to vector.

Fortunately, some excellent products can handle most of the complexities of conversion for you. If you are lucky enough to be converting between the formats supported by the *pbmplus* (Portable Bitmap Utilities) package (a freely available set of programs developed by Jef Poskanzer that we've included on the CD-ROM), your job will be an easy one. Rather than converting explicitly from one graphics file format to another (for example, from PCX to Microsoft Windows Bitmap [BMP]), pbmplus converts any source format to a common format and then converts that common format to the desired destination format. pbmplus is most often used in UNIX emvironments, but the source code has also been compiled for other platforms. We also provide a number of other freely available conversion programs for Windows, MS-DOS, OS/2, Macintosh, and UNIX platforms on the CD-ROM.

Try these programs out and see which best suits your formats and applications. If you have a machine that is running MS-DOS or a variant of Microsoft Windows, consider buying HiJaak, an excellent commercial product developed by Inset Systems (now Quarterdeck) that converts to and from most of the common file formats. If you have a Macintosh or Power Mac, consider buying DeBabelizer.

Chapter 7, *Format Conversion*, contains a discussion of converting between types of graphics file formats.

Compressing Data

Throughout the articles included in Part Two of this book, you'll see references to methods of *data compression* or *data encoding*. Compression is the process used to reduce the physical size of a block of information. By compressing

Ex.1007 / Page 56 of 1158
APPLE INC.

graphics data, we're able to fit more information in a physical storage space. Because graphics images usually require a very large amount of storage space, compression is an important consideration for graphics file formats. Almost every graphics file format uses some compression method.

There are several ways to look at compression. We can talk about differences between physical and logical compression, symmetric and asymmetric compression, and lossless and lossy compression. These terms are described in detail in Chapter 9, *Data Compression.* That chapter also describes the most common methods of, or algorithms for, compression, which we mention here briefly:

- Pixel packing—Not a method of data compression per se, but an efficient way to store data in contiguous bytes of memory. This method is used by the Macintosh PICT format and other formats that are capable of storing multiple 1-, 2-, or 4-bit pixels per byte of memory or disk space.
- Run-length encoding (RLE)—A very common compression algorithm used by such bitmap formats as BMP, TIFF, and PCX to reduce the amount of redundant graphics data.
- Lempel-Ziv-Welch (LZW)—Used by GIF and TIFF, this algorithm is also a part of the v.42bis modem compression standard and of PostScript Level 2.
- CCITT encoding—A form of data compression used for facsimile transmission and standardized by the CCITT. One particular standard is based on the keyed compression scheme introduced by David Huffman and known widely as Huffman encoding.
- Joint Photographic Experts Group (JPEG)—A toolkit of compression methods used particularly for continuous-tone image data and multimedia. The baseline JPEG implementation uses an encoding scheme based on the Discrete Cosine Transform (DCT) algorithm.
- Joint Bi-level Image Experts Group (JBIG)—A method of compressing bi-level (two-color) image data, which is intended to replace the MR (Modified READ) and MMR (Modified Modified READ) compression algorithms used by CCITT Group 3 and Group 4.
- ART—A proprietary compression algorithm developed by Johnson-Grace that can be adapted to support audio, animation, and full-motion video in the future.

Ex.1007 / Page 57 of 1158
APPLE INC.

- Fractal—A mathematical process used to encode bitmaps containing a real-world image as a set of mathematical data that describes the fractal (similar, repeating patterns) properties of the image.

Each of the articles in Part Two, *Graphics File Formats*, lists the compression algorithms used for the particular graphics file format described.

Format Summary

Table 1-1 lists each of the graphics file formats that we describe in this book, along with an indication of what type of format it is (bitmap, vector, metafile, scene description, animation, multimedia, or other [for "other" formats, refer to the appropriate article in Part Two]).

In some cases, a format may be known by a number of different names. Please check the index or look online if your format is not listed below; chances are good that it is included, but simply under a different name.

Graphics files on most platforms use a fairly consistent file extension convention. Of the three platforms with the largest installed base (MS-DOS, Macintosh, and UNIX), all use a similar *name.extension* filenaming convention. The other platforms that are popular for computer graphics (Amiga, Atari, and VMS) use a roughly similar naming convention. VMS, for example, uses the convention *name1.name2:version*, where *version* is an integer indicating the number of the file revision.

TABLE 1-1: ***Graphics File Formats Described in This Book***

Format	Type
Adobe Illustrator	Metafile
Adobe Photoshop	Bitmap
Atari ST Graphics Formats	Bitmap and Animation
AutoCAD DXF	Vector
Autodesk 3D Studio	Scene Description
BDF	Bitmap
BRL-CAD	Other
BUFR	Other
CALS Raster	Bitmap
CGM	Metafile
CMU Formats	Multimedia
DKB	Scene Description
Dore Raster File Format	Bitmap
DPX	Bitmap

Ex.1007 / Page 58 of 1158
APPLE INC.

Format	Type
Dr. Halo	Bitmap
DVM Movie	Animation
Encapsulated PostScript	Metafile (page description language)
FaceSaver	Bitmap
FAX Formats	Bitmap
FITS	Other
FLI	Animation
GEM Raster	Bitmap
GEM VDI	Metafile
GIF	Bitmap
GRASP	Animation
GRIB	Other
Harvard Graphics	Metafile
Hierarchical Data Format	Metafile
IFF	Bitmap
IGES	Other
Inset PIX	Bitmap
Intel DVI	Multimedia
JPEG File Interchange Format	Bitmap
Kodak Photo CD	Bitmap
Kodak YCC	Bitmap
Lotus DIF	Vector
Lotus PIC	Vector
Lumena Paint	Bitmap
Macintosh Paint	Bitmap
Macintosh PICT	Metafile
Microsoft Paint	Bitmap
Microsoft RIFF	Multimedia
Microsoft RTF	Metafile
Microsoft SYLK	Vector
Microsoft Windows Bitmap	Bitmap
Microsoft Windows Metafile	Metafile
MIFF	Bitmap
MPEG	Other
MTV	Scene Description
NAPLPS	Metafile
NFF	Scene Description
OFF	Scene Description
OS/2 Bitmap	Bitmap
P3D	Scene Description
PBM, PGM, PNM, and PPM	Bitmap
PCX	Bitmap
PDS	Other

Ex.1007 / Page 59 of 1158
APPLE INC.

Format	Type
Pictor PC Paint	Bitmap
Pixar RIB	Scene Description
Plot-10	Vector
PNG	Bitmap
POV	Vector
Presentation Manager Metafile	Metafile
PRT	Scene Description
QRT	Scene Description
QuickTime	Other
Radiance	Scene Description
Rayshade	Scene Description
RIX	Bitmap
RTrace	Scene Description
SAF	Bitmap and other
Sense8 NFF	Scene Description
SGI Image File Format	Bitmap
SGI Inventor	Scene Description
SGI YAODL	Scene Description
SGO	Vector
SPIFF	Bitmap
Sun Icon	Bitmap
Sun Raster	Bitmap
TDDD	Vector and Animation
TGA	Bitmap
TIFF	Bitmap
TTDDD	Vector and Animation
uRay	Scene Description
Utah RLE	Bitmap
VICAR2	Bitmap
VIFF	Bitmap
VIS-5D	Vector
Vivid and Bob	Scene Description
Wavefront OBJ	Vector
Wavefront RLA	Bitmap
WordPerfect Graphics Metafile	Metafile
XBM	Bitmap
XPM	Bitmap
XWD	Bitmap
ZBR	Metafile

Ex.1007 / Page 60 of 1158
APPLE INC.

CHAPTER 2

Computer Graphics Basics

To understand graphics file formats, you need some background in computer graphics. Of course, computer graphics is an enormous subject, and we can't hope to do it justice here. In general, we assume in this book that you are not a novice in this area. However, for those who do not have an extensive background in computer graphics, this chapter should be helpful in explaining the terminology you'll need to understand the discussions of the formats found later in this book.

If you're interested in exploring any of these topics further, far and away the best overall text is *Computer Graphics: Principles and Practice* by James D. Foley, Andries van Dam, S.K. Feiner, and J.F. Hughes. This is the second edition of the book formerly known throughout the industry as "Foley and van Dam." You'll find additional references in the "For Further Information" section at the end of this chapter.

Pixels and Coordinates

Locations in computer graphics are stored as mathematical coordinates, but the display surface of an output device is an actual physical object. Thus, it's important to keep in mind the distinction between physical pixels and logical pixels.

Physical Pixels

Physical pixels are the actual dots displayed on an output device. Each one takes up a small amount of space on the surface of the device. Physical pixels are manipulated directly by the display hardware and form the smallest independently programmable physical elements on the display surface. That's the

Ex.1007 / Page 61 of 1158
APPLE INC.

ideal, anyway. In practice, however, the display hardware may juxtapose or overlay several smaller dots to form an individual pixel. This is true in the case of most analog color CRT devices, which use several differently colored dots to display what the eye, at a normal viewing distance, perceives as a single, uniformly colored pixel.

Because physical pixels cover a fixed area on the display surface, there are practical limits to how close together two adjacent pixels can be. Asking a piece of display hardware to provide too high a resolution—too many pixels on a given display surface—will create blurring and other deterioration of image quality if adjacent pixels overlap or collide.

Logical Pixels

In contrast to physical pixels, *logical pixels* are like mathematical points: they specify a location, but are assumed to occupy no area. Thus, the mapping between logical pixel values in the bitmap data and physical pixels on the screen must take into account the actual size and arrangement of the physical pixels. A dense and brightly colored bitmap, for example, may lose its vibrancy when displayed on too large a monitor, because the pixels must be spread out to cover the surface.

Figure 2-1 illustrates the difference between physical and logical pixels.

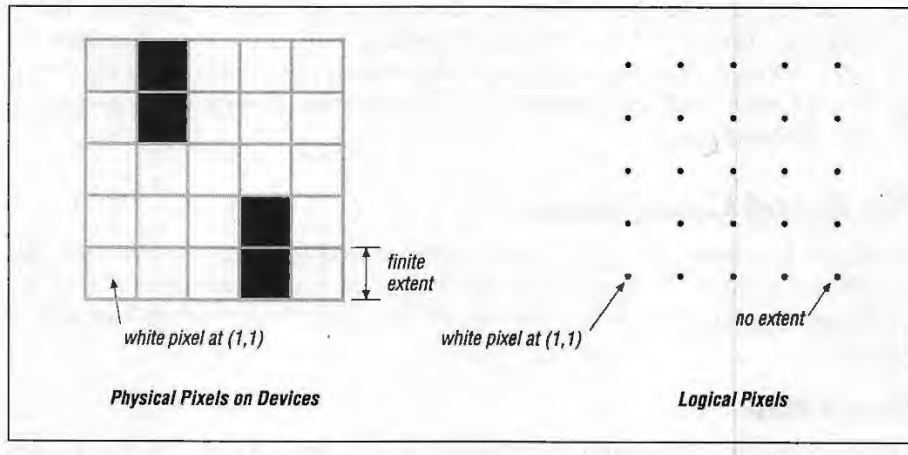


FIGURE 2-1: *Physical and logical pixels*

Ex.1007 / Page 62 of 1158
APPLE INC.

Pixel Depth and Displays

The number of bits in a value used to represent a pixel governs the number of colors the pixel can exhibit. The more bits per pixel, the greater the number of possible colors. More bits per pixel also means that more space is needed to store the pixel values representing a bitmap covering a given area on the surface of a display device. As technology has evolved, display devices handling more colors have become available at lower cost, which has fueled an increased demand for storage space.

Most modern output devices can display between two and more than 16 million colors simultaneously, corresponding to one and 24 bits of storage per pixel, respectively. *Bi-level*, or *1-bit*, displays use one bit of pixel-value information to represent each pixel, which then can have two color states. The most common 1-bit displays are monochrome monitors and black-and-white printers, of course. Things that reproduce well in black and white—line drawings, text, and some types of clip art—are usually stored as 1-bit data.

A Little Bit About Truecolor

People sometimes say that the human eye can discriminate between 2^{24} (16,777,216 colors), although many fewer colors can be perceived simultaneously. Naturally enough there is much disagreement about this figure, and the actual number certainly varies from person to person and under different conditions of illumination, health, genetics, and attention. In any case, we each can discriminate between a large number of colors, certainly more than a few thousand. A device capable of matching or exceeding the color-resolving power of the human eye under most conditions is said to display *truecolor*. In practice, this means 24 bits per pixel, but for historical reasons, output devices capable of displaying 2^{15} (32,768) or 2^{16} (65,536) colors have also incorrectly been called truecolor.

More recently, the term *hicolor* has come to be used for displays capable of handling up to 2^{15} or 2^{16} colors. *Fullcolor* is a term used primarily in marketing; its meaning is much less clear. (If you find out what it means, exactly, please let us know!)

Ex.1007 / Page 63 of 1158
APPLE INC.

Issues When Displaying Colors

It is frequently the case that the number or actual set of colors defined by the pixel values stored in a file differs from those that can be displayed on the surface of an output device. It is then up to the rendering application to translate between the colors defined in the file and those expected by the output device. There is generally no problem when the number of colors defined by the pixel values found in the file (source) is much less than the number that can be displayed on the output device (destination). The rendering application in this case is able to choose among the destination colors to provide a match for each source color. But a problem occurs when the number of colors defined by the pixel values exceeds the number that can be displayed on an output device. Consider the following examples.

In the first case, 4-bit-per-pixel data (corresponding to 16 colors) is being displayed on a device capable of supporting 24-bit data (corresponding to more than 16 million colors). The output device is capable of displaying substantially more colors than are needed to reproduce the image defined in the file. Thus, colors in the bitmap data will likely be represented by a close match on the output device, provided that the colors in the source bitmap and on the destination device are evenly distributed among all possible colors. Figure 2-2 illustrates this case.

This process, called *quantization*, results in a loss of data. For source images containing many colors, quantization can cause unacceptable changes in appearance, which are said to be the result of *quantization artifacts*. Examples of common quantization artifacts are banding, Moire patterns, and the introduction of new colors into the destination image that were not present in the source data. Quantization artifacts have their uses, however; one type of quantization process, called *convolution*, can be used to remove spurious noise from an image and to actually improve its appearance. On the other hand, it can also change the color balance of a destination image from that defined by the source data.

In the next case, the output device can display fewer colors than are defined by the source data. A common example is the display of 8-bit-per-pixel data (corresponding to 256 colors) on a device capable of displaying 4-bit data (corresponding to 16 colors). In this case, there may be colors defined in the bitmap which cannot be represented on the 4-bit display. Thus, the rendering application must work to match the colors in the source and destination. At some point in the color conversion process, the number of colors defined in the source data must be reduced to match the number available on the destination device. This reduction, or quantization, step is illustrated in Figure 2-3.

Ex.1007 / Page 64 of 1158
APPLE INC.

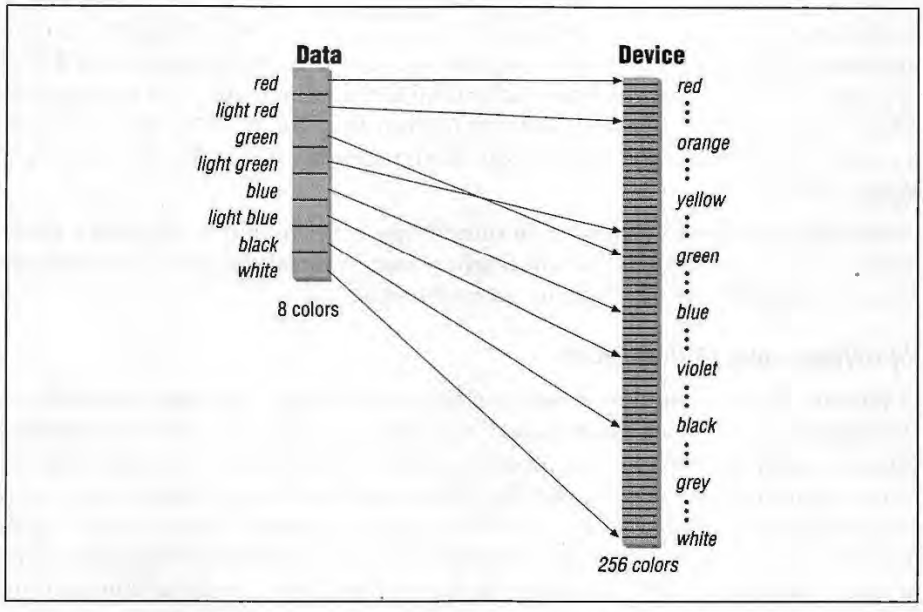


FIGURE 2-2: *Displaying data with few colors on a device with many colors*

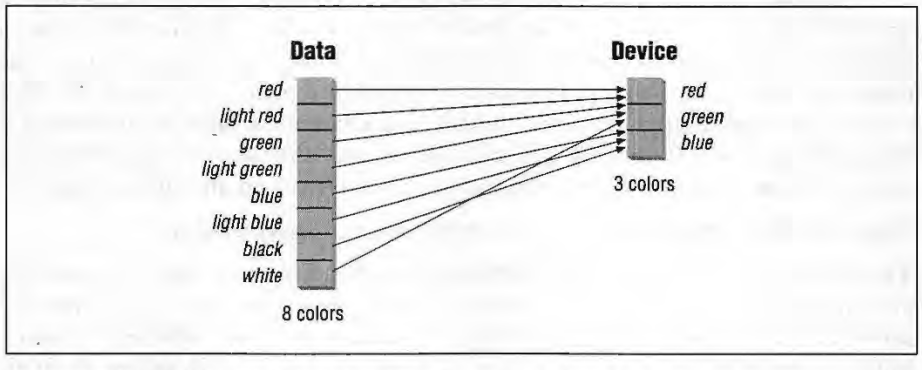


FIGURE 2-3: *Displaying data with many colors on a device with few colors*

Pixel Data and Palettes

Obviously, pixel values stored in a file correspond to colors. But how are the colors actually specified?

Ex.1007 / Page 65 of 1158
APPLE INC.

One-bit pixel data, capable of having the values 0 and 1, can only fully represent images containing two colors. Thus, there are only two ways of matching up pixel values in the file with colors on a screen. In most situations, you'll find that a convention already exists that establishes which value corresponds to which color, although a separate mechanism may be available in the file to change this. This definition can also be changed on the fly by the rendering application.

Pixel data consisting of more than one bit per pixel usually represents a set of *index values* into a *color palette*, although in some cases there is a direct numerical representation of the color in a *color definition scheme.*

Specifying Color With Palettes

A palette, which is sometimes referred to as a *color map, index map, color table,* or *look-up table (LUT)*, is a 1-dimensional array of color values. As the synonym *look-up table* suggests, it is the cornerstone of the method whereby colors can be referred to indirectly by specifying their positions in an array. Using this method, data in a file can be stored as a series of index values, usually small integer values, which can drastically reduce the size of the pixel data when only a small number of colors need to be represented. Bitmaps using this method of color representation are said to use *indirect,* or *pseudo-color* storage.

Four-bit pixel data, for instance, can be used to represent images consisting of 16 colors. These 16 colors are usually defined in a palette that is almost always included somewhere in the file. Each of the pixel values making up the pixel data is an index into this palette and consists of one of the values 0 to 15. The job of a rendering application is to read and examine a pixel value from the file, use it as an index into the palette, and retrieve the value of the color from the palette, which it then uses to specify a colored pixel on an output device.

Figure 2-4 illustrates how a palette may be used to specify a color.

The palette is an array of colors defined as accurately as possible. In practice, each palette element is usually 24 bits, or three bytes, long, although to accommodate future expansion and machine dependencies, each element is sometimes stored as 32 bits, or four bytes. Curiously, color models, many of which existed prior to the computer era, are often built around the equal partition of the possible colors into three variables, thus neatly fitting into three bytes of data storage. (We include a discussion of color models in the section called "Color" later in this chapter.)

Ex.1007 / Page 66 of 1158
APPLE INC.

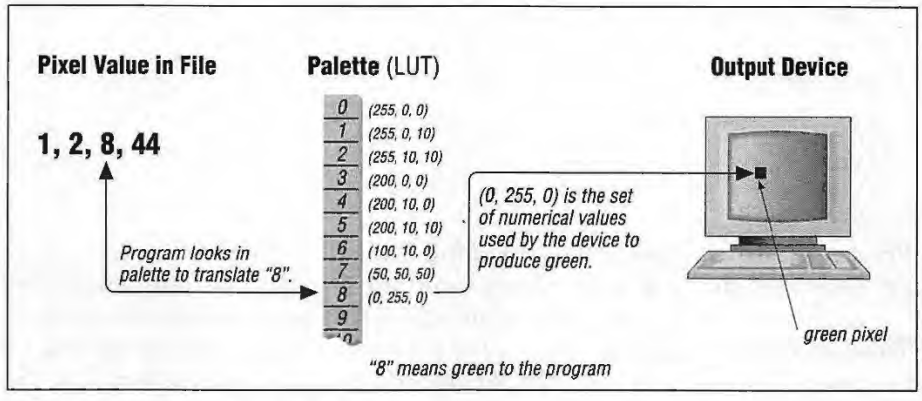


FIGURE 2-4: *Using a palette to specify a color*

What this means is that palettes are three or four times as large as the maximum number of colors defined. For instance, a 4-bit color palette is:

```
3 bytes per color * 16 colors = 48 bytes in length
```

or:

```
4 bytes per color * 16 colors = 64 bytes in length
```

depending on whether three or four bytes are used to store each color definition.

In a similar way, 8-bit pixel data may be used to represent images consisting of 256 colors. Each of the pixel values, in the range 0 to 255, is an index into a 256-color palette. In this case, the palette is:

```
3 bytes per color * 256 colors = 768 bytes in length
```

or:

```
4 bytes per color * 256 colors = 1024 bytes in length
```

Issues When Using Palettes

Let's say that the value (255,0,0) represents the color red in the color model used by our image format. We'll let our example palette define 16 colors, arranged as an array of 16 elements:

```
(  0,   0,   0)
(255,255,255)
(255,   0,   0)
(  0,255,   0)
(  0,   0,255)
(255,255,   0)
```


Ex.1007 / Page 67 of 1158
APPLE INC.

```
(  0,255,255)
(255,  0,255)
(128,  0,  0)
(  0,128,  0)
(  0,  0,128)
(128,128,  0)
(  0,128,128)
(128,  0,128)
(128,128,128)
(255,128,128)
```

Because (255,0,0) happens to be the third element in the palette, we can store the value 2 (if the array is zero-based, as in the C language), with the implied convention that the values are to be interpreted as index values into the array. Thus, every time a specification for the color red occurs in the pixel data, we can store 2 instead, and we can do likewise for other colors found in the image.

Color information can take up a substantial amount of space. In some cases, the use of palettes makes color storage more efficient; in other cases, storing colors directly, rather than through palettes, is more efficient.

In the larger, more complex image formats, indirect storage through the use of palettes saves space by reducing the amount of data stored in the file. If you are, for example, using a format that stores three bytes of color information per pixel (a commonly used method) and can use up to 256 colors, the pixel values making up the bitmap of a 320×200 pixel image would take up 192,000 (320 * 200 * 3) bytes of storage. If the same image instead used a palette with 256 3-byte elements, each pixel in the bitmap would only need to be one byte in size, just enough to hold a color map index value in the 0-to-255 range. This eliminates two of every three bytes in each pixel, reducing the needed storage to 64,000 (320 * 200 * 1) bytes.

Actually, we have to add in the length of the palette itself, which is 768 (256 * 3) bytes in length, so the relevant data in the file would be 64,768 bytes long, for a savings of nearly a factor of three over the former storage method. (Note, however, that if the amount of bitmap data in the file is very small, the storage overhead created by the inclusion of the palette may negate any savings gained by changing the storage method.)

Indirect color storage through the use of palettes has several advantages beyond the obvious. First, if you need to know how many actual colors are stored in an image (i.e., a 256-color image does not always contain 256 colors), it is a simple task to read through the palette and determine how many of its elements are being used or are duplicates of others. Unused elements in most formats are usually set to zero.

Ex.1007 / Page 68 of 1158
APPLE INC.

Palettes are also handy when you want to change the colors in an image. If you want to change all of the red pixels in the rendered image to green, for instance, all you need do is change the appropriate value defining the color red in the palette to the appropriate value for green.

As we've mentioned, the use of palettes is not appropriate in every case. A palette itself uses a substantial amount of space. For example, a palette defining 32,768 colors would take up a minimum of 98,304 bytes of storage space. For this reason, images containing more than 256 colors are generally stored in *literal*, *absolute*, or *truecolor* format (rather than in palettes), where each pixel value corresponds directly to a single color.

Palettes were devised to address the problem of the limited number of colors available on some display devices. However, if an output device does not provide hardware assistance to the application software, use of a palette-based format adds an extra level of complication prior to the appearance of the image on the display device. If the display device can support truecolor, it may be better to use a format supporting truecolor, even though the image may have only a few colors. As a general rule, images containing thousands or millions of colors are better stored using a format which supports truecolor, as the number and size of the elements needed in a palette-based format may cause the size of the palette needed to approach the size of the bitmapped image data itself.

Before we continue the discussion of how colors are stored in a file, we have to digress briefly to talk about how colors are defined. Discussion of palettes resumes in the section below called "... And Back to Palettes."

A Word About Color Spaces

Colors are defined by specifying several, usually three, values. These values specify the amount of each of a set of fundamental colors, sometimes called *color channels*, which are mixed to produce composite colors. A *composite color* is then specified as an ordered set of values. If "ordered set of values" rings a bell for you (in the same way as might "ordered pair"), rest assured that it also did for the people who create color definitions. A particular color is said to represent a point in a graphic plot of all the possible colors. Because of this, people sometimes refer to a color as a point in a *color space*.

RGB is a common color definition. In the RGB color model or system, the colors red, green, and blue are considered fundamental and undecomposable. A color can be specified by providing an RGB triplet in the form (R,G,B). People

Ex.1007 / Page 69 of 1158
APPLE INC.

sometimes think of color triplets in terms of percentages, although percentages are not, in fact, used to express actual color definitions. You might characterize colors in the RGB color model as follows:

(0%, 0%, 0%)	Black
(100%, 100%, 100%)	White
(100%, 0%, 0%)	Red
(50%, 50%, 50%)	Light gray

and so on.

There are many refinements of this, and you can always find somebody to argue about what numbers specify which color. This is the basic idea, though. Each of these RGB triplets is said to define a point in the RGB color space.

When storing color data in a file, it's more practical to specify the value of each color component, not as a percentage, but as a value in a predefined range. If the space allotted for each color component is a byte (eight bits), the natural range is 0 to 255. Because colors are commonly defined using 24 bits, or three bytes, the natural thing to do is to assign each of the three bytes for use as the value of the color component in the color model. In RGB color, for instance, using three bytes for each color, colors are usually stored as RGB triplets in the range 0 to 255, with 0 representing zero intensity and 255 representing maximum intensity.

```
RGB = ([0-255], [0-255], [0-255])
```

Thus, the pixel values in the previous example would be:

(0,0,0)	Black
(255,255,255)	White
(255,0,0)	Red
(127,127,127)	Light gray

This example assumes, of course, that 0 stands for the least amount, and 255 for the most amount of a particular color component. Occasionally, you will find that a format creator or application architect has perversely chosen to invert the "natural" sense of the color definition, and has made RGB (0, 0, 0) white and RGB (255, 255, 255) black, but, fortunately, this is rare.

The section later in this chapter called "How Colors are Represented" describes RGB and other color systems.

Ex.1007 / Page 70 of 1158
APPLE INC.

Some More About Truecolor

The word *truecolor* comes up in discussions about images that contain a large number of colors. What do we mean by large in this context? Most people consider 200 to 300K to be significantly large. Recall from the discussion above that a palette containing 256 color definitions uses a maximum of 64 bytes of storage, and that a palette with 32,768 or more colors uses nearly 100K, at a minimum. In light of this, 256 is not a "large" number of colors. Most people consider 32,768, 65,536, and 16.7 million colors to be "large," however. And this is only the space a palette takes up; we're not even talking about the image data!

Instead of including in a file a huge palette in which pixel values are indices into the palette, pixel values can be treated as literal color values. In practice, pixel values are composed of three parts, and each part represents a component color in the *color model* (e.g., RGB) in use. Pixel values from images containing 32,768 or 65,536 colors are typically stored in two successive bytes, or 16 bits, in the file, because almost all machines handle data a minimum of one byte at a time. A rendering application must read these 16-bit pixel values and decompose them into 5-bit color component values:

```
16 bits = 2 bytes = (8 bits, 8 bits) ⇒ (1, 5, 5, 5) = (1, R, G, B)
```

Each 5-bit component can have values in the range of 0 to 32. In the case of 32,768-color RGB images, only 15 bits are significant, and one bit is wasted or used for some other purpose. 65,536-color RGB images decompose the 16-bit pixel value asymmetrically, as shown below, in order to get use out of the extra bit:

```
16 bits = 2 bytes = (8 bits, 8 bits) ⇒ (6, 5, 5) = (R, G, B)
```

Actually, a more common subdivision is:

```
16 bits = 2 bytes = (8 bits, 8 bits) ⇒ (5, 6, 5) = (R, G, B)
```

Here, the extra bit is given to the green component, because the human eye is more sensitive to green than it is to red and blue. The color component order is arbitrary, and the order and interpretation of the color components within a pixel value varies from format to format. Thus, components of a 16-bit pixel value may be interpreted as (G,B,R) just as readily as (R,G,B) and (B,R,G). Specifying RGB colors in the sequence (R,G,B) has some appeal, because the colors are arranged by electromagnetic frequency, establishing their order in the physical spectrum.

Ex.1007 / Page 71 of 1158
APPLE INC.

24-bit pixel values are stored in either three bytes:

```
24 bits = 3 bytes = (8 bits, 8 bits, 8 bits) = (R,G,B)
```

or four bytes:

```
24 bits = 4 bytes = (8 bits, 8 bits, 8 bits, 8 bits) = (R,G,B, unused)
```

Equal division among the color components of the model, one byte to each component, is the most common scheme, although other divisions are not unheard of.

. . . And Back to Palettes

Earlier in this chapter we introduced the use of palettes. Here, we continue with the discussion of different types of palettes and illustrate with some actual examples.

Types of palettes

There are several different ways to talk about palettes.

A *single-channel palette* contains only one color value per element, and this color value maps directly to a single pixel color. Each element of a single-channel palette might have the following form, for example:

```
(G) = (223)
```

A *multiple-channel palette* (or *multi-channel palette)* contains two or more individual color values per color element. Each element of a 3-channel palette using red, green, and blue might have the following form, for example:

```
(R,G,B) = (255,128,78)
```

Here, R specifies the value of one channel, G specifies the value of the second channel, and B specifies the value of the third channel. If an image contains four color components, as with the CMYK color system described later in this chapter, then a 4-channel color map might be used, and so on.

Pixel-oriented palettes store all of the pixel color data as contiguous bits within each element of the array. As we noted above, in an RGB palette, each element in the palette consists of a triplet of values. This corresponds to the way pixel values are stored in the file, which is usually in RGB or BGR order:

```
(RGBRGBRGBRGBRGB...) or (BGRBGRBGRBGRBGR...)
```

Thus the palette looks like this:

```
(RGB)(RGB)(RGB) or (BGR)(BGR)(BGR)
```

In a plane-oriented palette, pixel color components are segregated; corresponding color-channel values are stored together, and the palette looks like it

Ex.1007 / Page 72 of 1158
APPLE INC.

is made up of three single-channel palettes, one for each color channel. This corresponds to the way pixel values are arranged in the file (i.e., as multiple color planes):

```
(RRRRR...GGGGG...BBBBB) or (BBBBB...GGGGG...RRRRR)
```

Thus, a small palette might look like this:

```
(R) (R) (R) (G) (G) (G) (B) (B) (B)
```

or:

```
(B) (B) (B) (G) (G) (G) (R) (R) (R)
```

Although this may look like a single palette containing three color planes, it is usually best to visualize it as three separate palettes, each containing a single color plane. This way you will have no trouble calling the first item in each color plane element zero.

It should be clear from the above discussion that both single- and multi-channel palettes can be pixel- or plane-oriented. For instance:

- A single-channel pixel-oriented palette contains one pixel value per element.
- A multi-channel pixel-oriented palette also contains one pixel per element, but each pixel contains two or more color channels of data.
- A single-channel plane-oriented palette contains one pixel per index and one bit per plane.
- A multi-channel plane-oriented palette contains one color channel value per element.

Figure 2-5 illustrates these different types of palettes.

As noted above, the number of elements in a palette is usually a power of two and typically corresponds to the maximum number of colors contained in the image, which is in turn reflected in the size of the pixel value in the file. For example, an 8-bit pixel value can represent 256 different colors and is accompanied by a 256-element palette. If an image has fewer colors than the maximum size of the palette, any unused elements in the palette will ideally be set to zero. Several formats, most notably CGM and TGA, have the ability to vary the number of elements in the palette as needed. If a TGA image contains only 57 colors, for instance, it may have only a 57-element palette.

It is also interesting to note that the usable elements in a palette are not always contiguously arranged, are not always ordered, and do not always start with the zero index value filled. A 2-color image with a 256-color palette (yes, it's been done) may have its colors indexed at locations 0 and 1, 0 and 255, 254 and

Ex.1007 / Page 73 of 1158
APPLE INC.

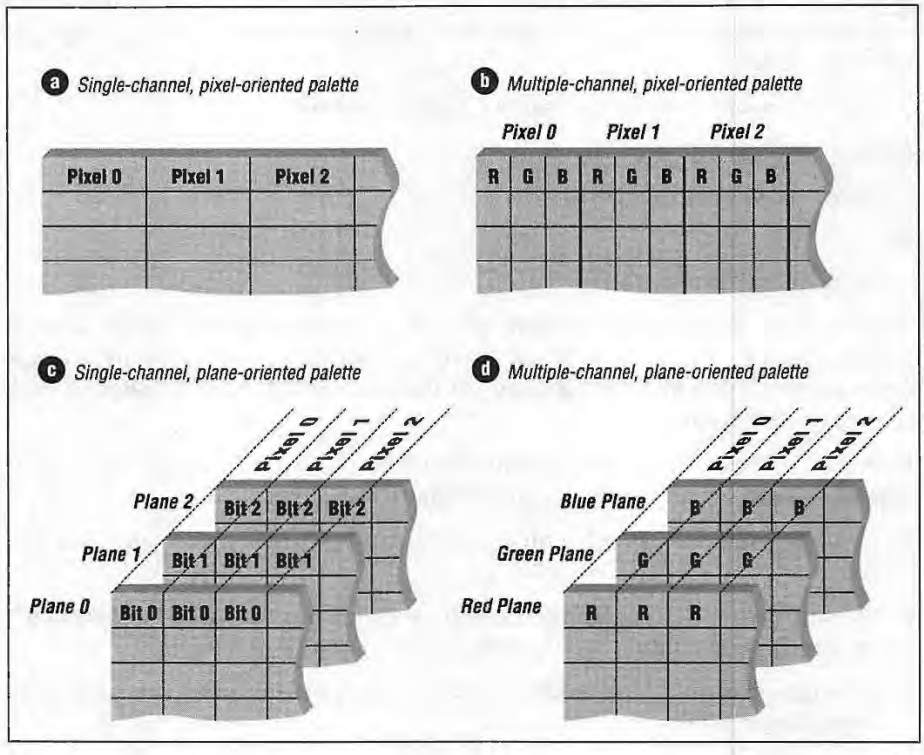


FIGURE 2-5: *Types of palettes*

255, or even 47 and 156. The locations are determined by the software writing the image file and therefore ultimately by the programmer who created the software application. (We choose not to comment further.)

Examples of palettes

Let's look at a few examples of palettes. The simplest is the 2-color, or monochrome, palette:

```
/* A BYTE is an 8-bit character */
typedef struct _MonoPalette
{
    BYTE Color[2];
} MONO_PALETTE;

MONO_PALETTE Mono = { {0x00, 0x01} };
```


Ex.1007 / Page 74 of 1158
APPLE INC.

In this example, we see a 2-element array containing the color values 0x00 and 0x01 in elements 0 and 1 respectively. In the file, all pixel values are indices. A pixel with a value of 0 serves as an index to the color represented by the value 0x00. Likewise, a pixel with a value of 1 serves as an index to the color represented by the value 0x01. Because this bitmap contains only two colors, and each pixel color may be represented by a single bit, it may seem easier to store these values directly in the bitmap as bit values rather than use a palette. It is easier, of course, but some palette-only formats require that this type of palette be present even for monochrome bitmaps.

This is a more practical example, a 16-element palette used to map a gray-scale palette:

```
/* A BYTE is an 8-bit character */
typedef struct _GrayPalette
{
    BYTE Color[16];
} GRAY_PALETTE;
GRAY_PALETTE Gray =
{
    {0x00,
    0x14,
    0x20,
    0x2c,
    0x38,
    0x45,
    0x51,
    0x61,
    0x71,
    0x82,
    0x92,
    0x92,
    0xa2,
    0xb6,
    0xcb,
    0xe3,
    0xff}
};
```

Notice in these two examples that each color element is represented by a single value, so this is a single-channel palette. We could just as easily use a 3-channel palette, representing each gray color element by its RGB value.

```
typedef struct _RGB
{
    BYTE    Red;    /* Red channel value */
    BYTE    Green;  /* Green channel value */
    BYTE    Blue;   /* Blue channel value */
} RGB;
```


Ex.1007 / Page 75 of 1158
APPLE INC.

```
RGB Gray[16] =
{
    {0x00, 0x00, 0x00},
    {0x14, 0x14, 0x14},
    {0x20, 0x20, 0x20},
    {0x2c, 0x2c, 0x2c},
    {0x38, 0x38, 0x38},
    {0x45, 0x45, 0x45},
    {0x51, 0x51, 0x51},
    {0x61, 0x61, 0x61},
    {0x71, 0x71, 0x71},
    {0x82, 0x82, 0x82},
    {0x92, 0x92, 0x92},
    {0xa2, 0xa2, 0xa2},
    {0xb6, 0xb6, 0xb6},
    {0xcb, 0xcb, 0xcb},
    {0xe3, 0xe3, 0xe3},
    {0xff, 0xff, 0xff}
};
```

This last example is an example of a pixel-oriented multi-channel palette. We can alter it to store the color information in a plane-oriented fashion like this:

```
TYPEDEF struct _PlanePalette
{
    BYTE    Red[16];    /* Red plane values */
    BYTE    Green[16];  /* Green plane values */
    BYTE    Blue[16];   /* Blue plane values */
} PLANE_PALETTE;

PLANE_PALETTE Planes =
{
    {0x00, 0x14, 0x20, 0x2c, 0x38, 0x45, 0x51, 0x61, /* Red plane */
    0x71, 0x82, 0x92, 0xa2, 0xb6, 0xcb, 0xe3, 0xff},
    {0x00, 0x14, 0x20, 0x2c, 0x38, 0x45, 0x51, 0x61, /* Green plane */
    0x71, 0x82, 0x92, 0xa2, 0xb6, 0xcb, 0xe3, 0xff},
    {0x00, 0x14, 0x20, 0x2c, 0x38, 0x45, 0x51, 0x61, /* Blue plane */
    0x71, 0x82, 0x92, 0xa2, 0xb6, 0xcb, 0xe3, 0xff}
};
```

Finally, let's look at a real-world example, the IBM VGA palette in wide use. This 256-color palette contains a 16-color sub-palette (the "EGA palette"), a 16-element gray-scale palette, and a palette of 24 colors, each with nine different variations of saturation and intensity. Notice that the last eight elements of the palette are not used and are thus set to zero:

```
struct _VgaPalette
{
    BYTE    Red;
    BYTE    Green;
    BYTE    Blue;
} VGA_PALETTE;
```


Ex.1007 / Page 76 of 1158
APPLE INC.

```
VGA_PALETTE VgaColors[256] =
{
/* EGA Color Table */
{0x00, 0x00, 0x00}, {0x00, 0x00, 0xaa},
{0x00, 0xaa, 0x00}, {0x00, 0xaa, 0xaa},
{0xaa, 0x00, 0x00}, {0xaa, 0x00, 0xaa},
{0xaa, 0x55, 0x00}, {0xaa, 0xaa, 0xaa},
{0x55, 0x55, 0x55}, {0x55, 0x55, 0xff},
{0x55, 0xff, 0x55}, {0x55, 0xff, 0xff},
{0xff, 0x55, 0x55}, {0xff, 0x55, 0xff},
{0xff, 0xff, 0x55}, {0xff, 0xff, 0xff},

/* Gray Scale Table */
{0x00, 0x00, 0x00}, {0x14, 0x14, 0x14},
{0x20, 0x20, 0x20}, {0x2c, 0x2c, 0x2c},
{0x38, 0x38, 0x38}, {0x45, 0x45, 0x45},
{0x51, 0x51, 0x51}, {0x61, 0x61, 0x61},
{0x71, 0x71, 0x71}, {0x82, 0x82, 0x82},
{0x92, 0x92, 0x92}, {0xa2, 0xa2, 0xa2},
{0xb6, 0xb6, 0xb6}, {0xcb, 0xcb, 0xcb},
{0xe3, 0xe3, 0xe3}, {0xff, 0xff, 0xff},

/* 24-color Table */
{0x00, 0x00, 0xff}, {0x41, 0x00, 0xff},
{0x7d, 0x00, 0xff}, {0xbe, 0x00, 0xff},
{0xff, 0x00, 0xff}, {0xff, 0x00, 0xbe},
{0xff, 0x00, 0x7d}, {0xff, 0x00, 0x41},
{0xff, 0x00, 0x00}, {0xff, 0x41, 0x00},
{0xff, 0x7d, 0x00}, {0xff, 0xbe, 0x00},
{0xff, 0xff, 0x00}, {0xbe, 0xff, 0x00},
{0x7d, 0xff, 0x00}, {0x41, 0xff, 0x00},
{0x00, 0xff, 0x00}, {0x00, 0xff, 0x41},
{0x00, 0xff, 0x7d}, {0x00, 0xff, 0xbe},
{0x00, 0xff, 0xff}, {0x00, 0xbe, 0xff},
{0x00, 0x7d, 0xff}, {0x00, 0x41, 0xff},
{0x7d, 0x7d, 0xff}, {0x9e, 0x7d, 0xff},
{0xbe, 0x7d, 0xff}, {0xdf, 0x7d, 0xff},
{0xff, 0x7d, 0xff}, {0xff, 0x7d, 0xdf},
{0xff, 0x7d, 0xbe}, {0xff, 0x7d, 0x9e},
{0xff, 0x7d, 0x7d}, {0xff, 0x9e, 0x7d},
{0xff, 0xbe, 0x7d}, {0xff, 0xdf, 0x7d},
{0xff, 0xff, 0x7d}, {0xdf, 0xff, 0x7d},
{0xbe, 0xff, 0x7d}, {0x9e, 0xff, 0x7d},
{0x7d, 0xff, 0x7d}, {0x7d, 0xff, 0x9e},
{0x7d, 0xff, 0xbe}, {0x7d, 0xff, 0xdf},
{0x7d, 0xff, 0xff}, {0x7d, 0xdf, 0xff},
{0x7d, 0xbe, 0xff}, {0x7d, 0x9e, 0xff},
{0xb6, 0xb6, 0xff}, {0xc7, 0xb6, 0xff},
{0xdb, 0xb6, 0xff}, {0xeb, 0xb6, 0xff},
{0xff, 0xb6, 0xff}, {0xff, 0xb6, 0xeb},
{0xff, 0xb6, 0xdb}, {0xff, 0xb6, 0xc7},
{0xff, 0xb6, 0xb6}, {0xff, 0xc7, 0xb6},
{0xff, 0xdb, 0xb6}, {0xff, 0xeb, 0xb6},
{0xff, 0xff, 0xb6}, {0xeb, 0xff, 0xb6},
```


Ex.1007 / Page 77 of 1158
APPLE INC.

```
{0xdb, 0xff, 0xb6}, {0xc7, 0xff, 0xb6},
{0xb6, 0xdf, 0xb6}, {0xb6, 0xff, 0xc7},
{0xb6, 0xff, 0xdb}, {0xb6, 0xff, 0xeb},
{0xb6, 0xff, 0xff}, {0xb6, 0xeb, 0xff},
{0xb6, 0xdb, 0xff}, {0xb6, 0xc7, 0xff},
{0x00, 0x00, 0x71}, {0x1c, 0x00, 0x71},
{0x38, 0x00, 0x71}, {0x55, 0x00, 0x71},
{0x71, 0x00, 0x71}, {0x71, 0x00, 0x55},
{0x71, 0x00, 0x38}, {0x71, 0x00, 0x1c},
{0x71, 0x00, 0x00}, {0x71, 0x1c, 0x00},
{0x71, 0x38, 0x00}, {0x71, 0x55, 0x00},
{0x71, 0x71, 0x00}, {0x55, 0x71, 0x00},
{0x38, 0x71, 0x00}, {0x1c, 0x71, 0x00},
{0x00, 0x71, 0x00}, {0x00, 0x71, 0x1c},
{0x00, 0x71, 0x38}, {0x00, 0x71, 0x55},
{0x00, 0x71, 0x71}, {0x00, 0x55, 0x71},
{0x00, 0x38, 0x71}, {0x00, 0x1c, 0x71},
{0x38, 0x38, 0x71}, {0x45, 0x38, 0x71},
{0x55, 0x38, 0x71}, {0x61, 0x38, 0x71},
{0x71, 0x38, 0x71}, {0x71, 0x38, 0x61},
{0x71, 0x38, 0x55}, {0x71, 0x38, 0x45},
{0x71, 0x38, 0x38}, {0x71, 0x45, 0x38},
{0x71, 0x55, 0x38}, {0x71, 0x61, 0x38},
{0x71, 0x71, 0x38}, {0x61, 0x71, 0x38},
{0x55, 0x71, 0x38}, {0x45, 0x71, 0x38},
{0x38, 0x71, 0x38}, {0x38, 0x71, 0x45},
{0x38, 0x71, 0x55}, {0x38, 0x71, 0x61},
{0x38, 0x71, 0x71}, {0x38, 0x61, 0x71},
{0x38, 0x55, 0x71}, {0x38, 0x45, 0x71},
{0x51, 0x51, 0x71}, {0x59, 0x51, 0x71},
{0x61, 0x51, 0x71}, {0x69, 0x51, 0x71},
{0x71, 0x51, 0x71}, {0x71, 0x51, 0x69},
{0x71, 0x51, 0x61}, {0x71, 0x51, 0x59},
{0x71, 0x51, 0x51}, {0x71, 0x59, 0x51},
{0x71, 0x61, 0x51}, {0x71, 0x69, 0x51},
{0x71, 0x71, 0x51}, {0x69, 0x71, 0x51},
{0x61, 0x71, 0x51}, {0x59, 0x71, 0x51},
{0x51, 0x71, 0x51}, {0x51, 0x71, 0x59},
{0x51, 0x71, 0x61}, {0x51, 0x71, 0x69},
{0x51, 0x71, 0x71}, {0x51, 0x69, 0x71},
{0x51, 0x61, 0x71}, {0x51, 0x59, 0x71},
{0x00, 0x00, 0x41}, {0x10, 0x00, 0x41},
{0x20, 0x00, 0x41}, {0x30, 0x00, 0x41},
{0x41, 0x00, 0x41}, {0x41, 0x00, 0x30},
{0x41, 0x00, 0x20}, {0x41, 0x00, 0x10},
{0x41, 0x00, 0x00}, {0x41, 0x10, 0x00},
{0x41, 0x20, 0x00}, {0x41, 0x30, 0x00},
{0x41, 0x41, 0x00}, {0x30, 0x41, 0x00},
{0x20, 0x41, 0x00}, {0x10, 0x41, 0x00},
{0x00, 0x41, 0x00}, {0x00, 0x41, 0x10},
{0x00, 0x41, 0x20}, {0x00, 0x41, 0x30},
{0x00, 0x41, 0x41}, {0x00, 0x30, 0x41},
{0x00, 0x20, 0x41}, {0x00, 0x10, 0x41},
{0x20, 0x20, 0x41}, {0x28, 0x20, 0x41},
```


Ex.1007 / Page 78 of 1158
APPLE INC.

```
{0x30, 0x20, 0x41}, {0x38, 0x20, 0x41},
{0x41, 0x20, 0x41}, {0x41, 0x20, 0x38},
{0x41, 0x20, 0x30}, {0x41, 0x20, 0x28},
{0x41, 0x20, 0x20}, {0x41, 0x28, 0x20},
{0x41, 0x30, 0x20}, {0x41, 0x38, 0x20},
{0x41, 0x41, 0x20}, {0x38, 0x41, 0x20},
{0x30, 0x41, 0x20}, {0x28, 0x41, 0x20},
{0x20, 0x41, 0x20}, {0x20, 0x41, 0x28},
{0x20, 0x41, 0x30}, {0x20, 0x41, 0x38},
{0x20, 0x41, 0x41}, {0x20, 0x38, 0x41},
{0x20, 0x30, 0x41}, {0x20, 0x28, 0x41},
{0x2c, 0x2c, 0x41}, {0x30, 0x2c, 0x41},
{0x34, 0x2c, 0x41}, {0x3c, 0x2c, 0x41},
{0x41, 0x2c, 0x41}, {0x41, 0x2c, 0x3c},
{0x41, 0x2c, 0x34}, {0x41, 0x2c, 0x30},
{0x41, 0x2c, 0x2c}, {0x41, 0x30, 0x2c},
{0x41, 0x34, 0x2c}, {0x41, 0x3c, 0x2c},
{0x41, 0x41, 0x2c}, {0x3c, 0x41, 0x2c},
{0x34, 0x41, 0x2c}, {0x30, 0x41, 0x2c},
{0x2c, 0x41, 0x2c}, {0x2c, 0x41, 0x30},
{0x2c, 0x41, 0x34}, {0x2c, 0x41, 0x3c},
{0x2c, 0x41, 0x41}, {0x2c, 0x3c, 0x41},
{0x2c, 0x34, 0x41}, {0x2c, 0x30, 0x41},
{0x00, 0x00, 0x00}, {0x00, 0x00, 0x00},
{0x00, 0x00, 0x00}, {0x00, 0x00, 0x00},
{0x00, 0x00, 0x00}, {0x00, 0x00, 0x00},
{0x00, 0x00, 0x00}, {0x00, 0x00, 0x00}
};
```

Color

Understanding how colors are defined in graphics data is important to understanding graphics file formats. In this section, we touch on some of the many factors governing how colors are perceived. This is by no means a comprehensive discussion. We just want to make sure that you have an appreciation of some of the problems that come up when people start to deal with color.

How We See Color

The eye has a finite number of color receptors that, taken together, respond to the full range of light frequencies (about 380 to 770 nanometers). As a result, the eye theoretically supports only the perception of about 10,000 different colors simultaneously (although, as we have mentioned, many more colors than this can be perceived, though not resolved simultaneously).

The eye is also biased to the kind of light it detects. It's most sensitive to green light, followed by red, and then blue. It's also the case that the visual perception system can sense contrasts between adjacent colors more easily than it can

Ex.1007 / Page 79 of 1158
APPLE INC.

sense absolute color differences, particularly if those colors are physically separated in the object being viewed. In addition, the ability to discern colors varies from person to person; it's been estimated that one out of every twelve people has some form of color blindness.

Furthermore, the eye is limited in its ability to resolve the color of tiny objects. The size of a pixel on a typical CRT display screen, for example, is less than a third of a millimeter in diameter. When a large number of pixels are packed together, each one a different color, the eye is unable to resolve where one pixel ends and the next one begins from a normal viewing distance. The brain, however, must do something to bridge the gap between two adjacent differently colored pixels and will integrate, average, ignore the blur, or otherwise adapt to the situation. For these reasons and others, the eye typically perceives many fewer colors than are physically displayed on the output device.

How a color is created also plays an important role in how it is perceived. Normally, we think of colors as being associated with a single wavelength of light. We know, however, that two or more colors can be mixed together to produce a new color. An example of this is mixing green and red light to produce yellow light, or mixing yellow and blue pigments to produce green pigment. This mixing of colors can also occur when an object is illuminated by light. The color of the object will always mix with the color of the light to produce a third color. A blue object illuminated by white light appears blue, while the same object illuminated by red light will appear violet in color.

One implication of this is that the same image rendered to two different devices will look different. Two different color monitors, for example, seldom produce identically perceived images, even if the monitors are the same make and model. Another implication is that images rendered to different *types* of devices will look different. An example is the difference between an image rendered to a monitor and one rendered to a color hardcopy device. Although there are numerous schemes designed to minimize color-matching problems, none is wholly satisfactory.

For these and other reasons, the accurate rendition of color is full of difficulties, and much work continues to be done. Although a number of mechanical devices have recently appeared on the market, they are for the most part designed to work with one type of output device. The ultimate arbiter of color quality will always be the person who views the image on the output device.

Ex.1007 / Page 80 of 1158
APPLE INC.

How Colors Are Represented

Several different mathematical systems exist which are used to describe colors. This section describes briefly the color systems most commonly used in the graphics file formats described in this book.

NOTE

> Keep in mind that perception of color is affected by physiology, experience, and viewing conditions. For these reasons, no system of color representation has yet been defined, or indeed is likely to be, which is satisfactory under all circumstances.

For purposes of discussion here, colors are always represented by numerical values. The most appropriate color system to use depends upon the type of data contained in the file. For example, 1-bit, gray-scale, and color data might each best be stored using a different color model.

Color systems used in graphics files are typically of the *trichromatic colorimetric* variety, otherwise known as *primary 3-color systems.* With such systems, a color is defined by specifying an ordered set of three values. Composite colors are created by mixing varying amounts of three *primary colors,* which results in the creation of a new color. Primary colors are those which cannot be created by mixing other colors. The totality of colors that can be created by mixing primary colors make up the color space or *color gamut.*

Additive and subtractive color systems

Color systems can be separated into two categories: *additive* color systems and *subtractive* color systems. Colors in additive systems are created by adding colors to black to create new colors. The more color that is added, the more the resulting color tends towards white. The presence of all the primary colors in sufficient amounts creates pure white, while the absence of all the primary colors creates pure black. Additive color environments are self-luminous. Color on monitors, for instance, is additive.

Color subtraction works in the opposite way. Conceptually, primary colors are subtracted from white to create new colors. The more color that is subtracted, the more the resulting color tends towards black. Thus, the presence of all the primary colors theoretically creates pure black, while the absence of all primary colors theoretically creates pure white. Another way of looking at this process is that black is the total absorption of all light by color pigments. Subtractive environments are reflective in nature, and color is conveyed to us by reflecting light from an external source. Any color image reproduced on paper is an example of the use of a subtractive color system.

Ex.1007 / Page 81 of 1158
APPLE INC.

No color system is perfect. As an example, in a subtractive color system the presence of all colors creates black, but in real-life printing the inks are not perfect. Mixing all ink colors usually produces a muddy brown rather than black. The blacks we see on paper are only approximations of the mathematical ideal, and likewise for other colors.

The next few sections describe some common color systems. Table 2-1 shows corresponding values for the primary and achromatic colors using the RGB, CMY, and HSV color systems.

RGB (Red-Green-Blue)

RGB is perhaps the most widely used color system in image formats today. It is an additive system in which varying amounts of the colors red, green, and blue are added to black to produce new colors. Graphics files using the RGB color system represent each pixel as a color triplet—three numerical values in the form (R,G,B), each representing the amount of red, green, and blue in the pixel, respectively. For 24-bit color, the triplet (0,0,0) normally represents black, and the triplet (255,255,255) represents white. When the three RGB values are set to the same value—for example, (63,63,63) or (127,127,127), or (191,191,191)—the resulting color is a shade of gray.

TABLE 2-1: ***Equivalent RGB, CMY, and HSV values***

	RGB	CMY	HSV
Red	255,0,0	0,255,255	0,240,120
Yellow	255,255,0	0,0,255	40,240,120
Green	0,255,0	255,0,255	80,240,120
Cyan	0,255,255	255,0,0	120,240,120
Blue	0,0,255	255,255,0	160,240,120
Magenta	255,0,255	0,255,0	200,240,120
Black	0,0,0	255,255,255	160,0,0
Shades of Gray	63,63,63	191,191,191	160,0,59
of	127,127,127	127,127,127	160,0,120
Gray	191,191,191	63,63,63	160,0,180
White	255,255,255	0,0,0	160,0,240

Ex.1007 / Page 82 of 1158
APPLE INC.

CMY (Cyan-Magenta-Yellow)

CMY is a subtractive color system used by printers and photographers for the rendering of colors with ink or emulsion, normally on a white surface. It is used by most hard-copy devices that deposit color pigments on white paper, such as laser and ink-jet printers. When illuminated, each of the three colors absorbs its complementary light color. Cyan absorbs red; magenta absorbs green; and yellow absorbs blue. By increasing the amount of yellow ink, for instance, the amount of blue in the image is decreased.

As in all subtractive systems, we say that in the CMY system colors are subtracted from white light by pigments to create new colors. The new colors are the wavelengths of light reflected, rather than absorbed, by the CMY pigments. For example, when cyan and magenta are absorbed, the resulting color is yellow. The yellow pigment is said to "subtract" the cyan and magenta components from the reflected light. When all of the CMY components are subtracted, or absorbed, the resulting color is black. Almost. Whether it's possible to get a perfect black is debatable. Certainly, a good black color is not obtainable without expensive inks.

In light of this, the CMY system has spawned a practical variant, CMYK, with K standing for the color black. To compensate for inexpensive and off-specification inks, the color black is tacked onto the color system and treated something like an independent primary color variable. For this reason, use of the CMYK scheme is often called 4-color printing, or process color. In many systems, a dot of composite color is actually a grouping of four dots, each one of the CMYK colors. This can be readily seen with a magnifying lens by examining a color photograph reproduced in a glossy magazine.

CMYK can be represented as either a color triple, like RGB, or as four values. If expressed as a color triple, the individual color values are just the opposite of RGB. For a 24-bit pixel value, for example, the triplet (255,255,255) is black, and the triplet (0,0,0) is white. In most cases, however, CMYK is expressed as a series of four values.

In many real-world color composition systems, the four CMYK color components are specified as percentages in the range of 0 to 100.

HSV (Hue, Saturation, and Value)

HSV is one of many color systems that vary the degree of properties of colors to create new colors, rather than using a mixture of the colors themselves. Hue specifies "color" in the common use of the term, such as red, orange, blue, and so on. Saturation (also called chroma) refers to the amount of white in a hue;

Ex.1007 / Page 83 of 1158
APPLE INC.

a fully (100 percent) saturated hue contains no white and appears pure. By extension, a partly saturated hue appears lighter in color due to the admixture of white. Red hue with 50 percent saturation appears pink, for instance. Value (also called brightness) is the degree of self-luminescence of a color—that is, how much light it emits. A hue with high intensity is very bright, while a hue with low intensity is dark.

HSV (also called HSB for Hue, Saturation, and Brightness) most closely resembles the color system used by painters and other artists, who create colors by adding white, black, and gray to pure pigments to create tints, shades, and tones. A tint is a pure, fully saturated color combined with white, and a shade is a fully saturated color combined with black. A tone is a fully saturated color with both black and white (gray) added to it. If we relate HSV to this color mixing model, saturation is the amount of white, value is the amount of black, and hue is the color that the black and white are added to.

The HLS (Hue, Lightness, and Saturation) color model is closely related to HSV and behaves in the same way.

There are several other color systems that are similar to HSV in that they create color by altering hue with two other values. These include:

- HSI (Hue, Saturation, and Intensity)
- HSL (Hue, Saturation, and Luminosity)
- HBL (Hue, Brightness, and Luminosity)

None of these is widely used in graphics files.

YUV (Y-signal, U-signal, and V-signal)

The YUV model is a bit different from the other colorimetric models. It is basically a linear transformation of RGB image data and is most widely used to encode color for use in television transmission. (Note, however, that this transformation is almost always accompanied by a separate quantization operation, which introduces nonlinearities into the conversion.) Y specifies gray scale or luminance. The U and V components correspond to the chrominance (color information). Other color models based on YUV include YCbCr and YPbPr.

Black, White, and Gray

Black, white, and gray are considered neutral (achromatic) colors that have no hue or saturation. Black and white establish the extremes of the range, with black having minimum intensity, gray having intermediate intensity, and white having maximum intensity. One can say that the gamut of gray is just a specific slice of a color space, each of whose points contains an equal amount of the three primary colors, has no saturation, and varies only in intensity.

Ex.1007 / Page 84 of 1158
APPLE INC.

White, for convenience, is often treated in file format specifications as a primary color. Gray is usually treated the same as other composite colors. An 8-bit pixel value can represent 256 different composite colors or 256 different shades of gray. In 24-bit RGB color, (12,12,12), (128,128,128), and (199,199,199) are all shades of gray.

Overlays and Transparency

Certain file formats are designed to support the storage of still images captured from video sources. In practice, images of this sort are often overlaid on live video sources at render time. This is a familiar feature of conventional broadcast television, where still images are routinely shown next to live readers on the evening news.

Normal images are opaque, in the sense that no provision is made to allow the manipulation and display of multiple overlaid images. To allow image overlay, some mechanism must exist for the specification of transparency on a per-image, per-strip, per-tile, or per-pixel basis. In practice, transparency is usually controlled through the addition of information to each element of the pixel data.

The simplest way to allow image overlay is the addition of an *overlay bit* to each pixel value. Setting the overlay bit in an area of an image allows the rendering application or output device to selectively ignore those pixel values with the bit set. An example is the 16-bit variant of the TGA format, which supports data in the format:

```
(15 bits) = (R,G,B) = (5 bits, 5 bits, 5 bits)
```

Actually, this 15-bit pixel value is stored in 16 bits; an extra bit is left over which can be used to support the overlaying of images:

```
(R,G,B,T) = (16 bits) = (5 bits, 5 bits, 5 bits, 1 bit overlay)
```

The image creator or rendering application can toggle the overlay bit, which is interpreted by the display hardware as a command to ignore that particular pixel. In this way, two images can be overlaid, and the top one adjusted to allow holes through which portions of the bottom image are visible.

This technique is in widespread, but not obvious, use. A rendering application can selectively toggle the overlay bit in pixel values of a particular color. More to the point, the application can turn off the display of any area of an image that is not a particular color. For example, if a rendering application encounters an image of a person standing in front of a contrasting, uniformly colored

Ex.1007 / Page 85 of 1158
APPLE INC.

and lighted screen, the application can toggle the overlay bits on all the pixel values that are the color of the screen, leaving an image of the person cut out from the background. This cut-out image can then be overlaid on any other image, effectively adding the image of the person to the bottom image.

This assumes, of course, that the color of the screen is different from any colors in the person portion of the image. This is often how broadcast television weather reporters are overlaid on background maps and displays, for instance. Certain conventions inherited from traditional analog broadcasting technology are in widespread use in the broadcasting industry, including the use of a particular blue shade for background screens. When used in this way, the process is called *chromakeying.*

A more elaborate mechanism for specifying image overlays allows variations in transparency between bottom and overlaid images. Instead of having a single bit of overlay information, each pixel value has more (usually eight bits). An example is the 32-bit variant of the TGA format, which also supports data in the format:

```
(24 bits) = (R,G,B) = (8 bits, 8 bits, 8 bits)
```

Because this 24-bit pixel value is stored in 32 bits, an extra eight bits are left over to support transparency:

```
(R,G,B,T) = (8 bits, 8 bits, 8 bits, 8 bits transparency)
```

The eight transparency bits are sometimes called the *alpha channel.* Although there are some complications in the TGA format, an ideal 8-bit alpha channel can support 256 levels of transparency, from zero (indicating that the pixel is meant to be completely transparent) to 255 (indicating that the pixel is meant to be opaque).

Transparency data is usually stored as part of the pixel data, as in the example above, but it may also appear as a fourth plane, stored the same way as palette data in planar format files. It can, however, be stored as a separate block, independent of other image and palette information, and with the same dimensions as the actual image. This allows manipulation of the transparency data independent of the image pixel data.

For Further Information

In this chapter, we have been able to touch upon only a small part of the science of computer graphics and imaging. For additional information, see the references cited below.

Ex.1007 / Page 86 of 1158
APPLE INC.

Books About Computer Graphics

The following books are some of the best texts available on the subject of computer graphics. Many of them contain detailed chapters on specific application areas, such as imaging, ray tracing, animation, art, computer-aided design, and 3D modeling.

Artwick, Bruce A., *Applied Concepts in Microcomputer Graphics*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

Conrac Corporation, *Raster Graphics Handbook*, second edition, Van Nostrand Reinhold Company, New York, NY, 1985.

Foley, James. D., Andries van Dam, S.K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, second edition, Addison-Wesley, Reading, MA, 1990.

Hearn, Donald and M. Pauline Baker, *Computer Graphics*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Netravali, Arun N. and Barry G. Haskell, *Digital Pictures: Representation and Compression*, Plenum Press, New York, NY, 1988.

Newman, William N. and Robert F. Sproull, *Principles of Interactive Computer Graphics*, second edition, McGraw Hill, New York, NY, 1973.

Rogers, David F. and J. Alan Adams, *Mathematical Elements for Computer Graphics*, second edition, McGraw Hill, New York, NY, 1990.

Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw Hill, New York, NY, 1985.

Rosenfeld, Azriel and Avinash C. Kah, *Digital Picture Processing*, second edition, Academic Press, San Diego, CA, 1982.

Sharpe, L., "Tiling: Turning Unwieldy Drawings into Neat Little Packets," *Inform*, Association for Image and Information Management, March 1989.

Watt, Alan, *Fundamentals of Three-Dimensional Computer Graphics*, Addison-Wesley, Reading, MA, 1989.

Books About Color and Colorimetry

The following books are excellent reference works on color, its measurement, and its effects on the human psycho-biological system.

Ex.1007 / Page 87 of 1158
APPLE INC.

Benson, K. Blair, *Television Engineering Handbook*, second edition, McGraw Hill, New York, NY, 1986.

Billmeyer, Fred W. and Max Saltzman, *Principles of Color Technology*, second edition, John Wiley & Sons, New York, NY, 1981.

De Grandis, Luigina, *Theory and Use of Color*, translated by John Gilbert, Harry N. Abrams, Inc., New York, NY, 1986.

Foley, James. D., Andries van Dam, S.K. Feiner, and J.F. Hughes, *Computer Graphics: Principles and Practice*, second edition, Addison-Wesley, Reading, MA, 1990.

Hunt, R.W.G., *Measuring Color*, second edition, E. Horwood, New York, NY, 1991.

Hunt, R.W.G., *The Reproduction of Color*, third edition, John Wiley & Sons, New York, NY, 1975.

Hunt, R.W.G., *The Reproduction of Color in Photography, Printing and Television*, Fountain Press, Tolworth, England, 1987.

Judd, Deane B., *Color in Business, Science, and Industry*, third edition, John Wiley & Sons, New York, NY, 1975.

Kueppers, Harald, *Color; Origin, Systems, Uses*, translated by Bradley, F., Von Nostrand Reinhold Ltd., London, England, 1973.

Optical Society of America, Committee on Colorimetry, *The Science of Color*, Washington, DC, 1963.

Wyszecki, Gunter and W.S. Stiles, *Color Science: Concepts and Methods, Quantitative Data and Formulae*, second edition, John Wiley & Sons, New York, NY, 1982.

Ex.1007 / Page 88 of 1158
APPLE INC.

CHAPTER 3

Bitmap Files

Bitmap files vary greatly in their details, but they all share the same general structure. This chapter looks at the components of a typical bitmap file. Later on in this chapter we'll get into explanations of the details, but for now let's just get a feel for the overall structure. We'll explain as necessary as we go along.

Bitmap files consist of a header, bitmap data, and other information, which may include a color palette and other data.

A warning: inexplicably, people continue to design applications which use what are sometimes called raw formats. *Raw* format files consist solely of *image data* and omit any clues as to their structure. Both the creator of such files and the rendering applications must somehow know, ahead of time, how the files are structured. Because you usually can't tell one raw format file from another (except perhaps, by examining their relative sizes), we'll confine our discussion in this chapter to bitmap files, which at least contain *headers.*

How Bitmap Files Are Organized

The basic components of a simple bitmap file are the following:

Header
Bitmap Data

If a file contains no image data, only a header will be present. If additional information is required that does not fit in the header, a footer will usually be present as well:

Ex.1007 / Page 89 of 1158
APPLE INC.

Header
Bitmap Data
Footer

An image file may store a palette in the header, but it will more likely appear immediately after the header:

Header
Palette
Bitmap Data
Footer

A palette can also appear immediately after the image data, like a footer, or be stored in the footer itself:

Header
Bitmap Data
Palette

Scan-line tables and *color correction tables* may also appear after the header and before or after the image data:

Header
Palette
Scan Line Table
Color Correction Table (here)
Bitmap Data
Color Correction Table (or here)
Footer

If an image file format is capable of holding multiple images, then an *image file index* may appear after the header, holding the offset values of the starting positions of the images in the file:

Ex.1007 / Page 90 of 1158
APPLE INC.

Header
Palette
Bitmap Index
Bitmap 2 Data
...
Bitmap *n* Data
Footer

If the format definition allows each image to have its own palette, the palette will most likely appear before the image data with which it is associated:

Header
Palette
Bitmap Index
Palette 1
Bitmap Data
Palette 2
Bitmap 2 Data
...
Palette *n*
Bitmap *n* Data
Footer

We'll now look at the parts of a bitmap file piece by piece.

Header

The header is a section of *binary-* or *ASCII-format* data normally found at the beginning of the file, containing information about the bitmap data found elsewhere in the file. All bitmap files have some sort of header, although the format of the header and the information stored in it varies considerably from format to format. Typically, a bitmap header is composed of fixed fields. None of these fields is absolutely necessary, nor are they found in all formats, but this

Ex.1007 / Page 91 of 1158
APPLE INC.

list is typical of those formats in widespread use today. The following information is commonly found in a bitmap header:

Header
Palette
Bitmap Index
Palette 1
File Identifier
File Version
Number of Lines per Image
Number of Pixels per Line
Number of Bits per Pixel
Number of Color Planes
Compression Type
X Origin of Image
Y Origin of Image
Text Description
Unused Space

Later in this chapter we will present examples of headers from several actual formats, containing fields similar to those presented above.

File Identifier

A header usually starts with some sort of unique identification value called a *file identifier*, *file ID*, or *ID value*. Its purpose is to allow a software application to determine the format of the particular graphics file being accessed.

ID values are often *magic values* in the sense that they are assigned arbitrarily by the creator of the file format. They can be a series of ASCII characters, such as BM or GIF, or a 2- or 4-byte word value, such as 4242h or 596aa695h, or any other pattern that made sense to the format creator. The pattern is usually assumed to be unique, even across platforms, but this is not always the case, as we describe in the next few paragraphs. Usually, if a value in the right place in a file matches the expected identification value, the application reading the file header can assume that the format of the image file is known.

Ex.1007 / Page 92 of 1158
APPLE INC.

Three circumstances arise, however, which make this less than a hard and fast rule. Some formats omit the image file identifier, starting off with data that can change from file to file. In this case, there is a small probability that the data will accidentally duplicate one of the magic values of another file format known to the application. Fortunately, the chance of this occurring is remote.

The second circumstance can come about when a new format is created and the format creator inadvertently duplicates, in whole or in part, the magic values of another format. In case this seems even more unlikely than accidental duplication, rest assured that it has already happened several times. Probably the chief cause is that, historically, programmers have borrowed ideas from other platforms, secure in the belief that their efforts would be isolated behind the "Chinese Wall" of binary incompatibility. In the past, confusion of formats with similar ID fields seldom came about and was often resolved by context when it did happen. Obviously this naive approach by format creators is no longer a survival skill. In the future, we can expect more problems of this sort as users, through local area networking and through advances in regional and global interconnectivity, gain access to data created on other platforms.

This third circumstance comes about when a vendor—either the format creator or format caretaker or a third party—changes, intentionally or unintentionally, the specification of the format, while keeping the ID value specified in the format documentation. In this case, an application can recognize the format, but be unable to read some or all of the data. If the idea of a vendor creating intentional, undocumented changes seems unlikely, rest assured that this, too, has already happened many times. Examples are the GIF, TIFF, and TGA file formats. In the case of the GIF and TGA formats, vendors (not necessarily the format creators) have extended or altered the formats to include new data types. In the case of TIFF, vendors have created and promulgated what only can be described as *convenience revisions*, apparently designed to accommodate coding errors or application program quirks.

File Version

Following the identification value in the header is usually a field containing the file version. Naturally enough, successive versions of bitmap formats may differ in characteristics such as header size, bitmap data supported, and color capability. Once having verified the file format through the ID value, an application will typically examine the version value to determine if it can handle the image data contained in the file.

Ex.1007 / Page 93 of 1158
APPLE INC.

Image Description Information

Next comes a series of fields that describe the image itself. As we will see, bitmaps are usually organized, either physically or logically, into lines of pixels. The field designated *number of lines per image*, also called the *image length*, *image height*, or *number of scan lines*, holds a value corresponding to the number of lines making up the actual bitmap data. The *number of pixels per line*, also called the *image width* or *scan-line width*, indicates the number of pixels stored in each line.

The number of bits per pixel indicates the size of the data needed to describe each pixel per *color plane*. This may also be stored as the *number of bytes per pixel*, and is more properly called *pixel depth*. Forgetting the exact interpretation of this field when coding format readers is a common source of error. If the bitmap data is stored in a series of planes, the *number of color planes* indicates the number of planes used. Often the value defaults to one. There is an increasing tendency to store bitmaps in single-plane format, but multi-plane formats continue to be used in support of special hardware and alternate color models.

The number of bits in a line of the image can be calculated by multiplying the values of *number of bits per pixel*, *number of pixels per line*, and *number of color planes* together. We can determine the number of bytes per scan line by then dividing the resulting product by eight. Note that there is nothing requiring *number of bits per pixel* to be an integral number of 8-bit bytes.

Compression Type

If the format supports some sort of *encoding* designed to reduce the size of the bitmap data, then a *compression type* field will be found in the header. Some formats support multiple compression types, including *raw* or *uncompressed* data. Some format revisions consist mainly of additions or changes to the compression scheme used. Data compression is an active field, and new types of compression accommodating advances in technology appear with some regularity. TIFF is one of the common formats which has exhibited this pattern in the past.

For more information about compression, see Chapter 9, *Data Compression.*

x and y Origins

x origin of image and *y origin of image* specify a coordinate pair that indicates where the image starts on the output device. The most common origin pair is 0,0, which puts one corner of the image at the origin point of the device. Changing these values normally causes the image to be displayed at a different location when it is rendered.

Ex.1007 / Page 94 of 1158
APPLE INC.

Most bitmap formats were designed with certain assumptions about the output device in mind, and thus can be said to model either an actual or virtual device having a feature called the *drawing surface.* The drawing surface has an implied origin, which defines the starting point of the image, and an implied orientation, which defines the direction in which successive lines are drawn as the output image is rendered. Various formats and display devices vary in the positioning of the origin point and orientation direction. Many place the origin in the upper-left corner of the display surface, although it can also appear in the center, or in the lower-left corner. Others, although this is far less common, put it in the upper- or lower-right corner.

Orientation models with the origin in the upper-left corner are often said to have been created in support of hardware, and there may be some historical and real-world justification for this. People with backgrounds in mathematics and the natural sciences, however, are used to having the origin in the lower-left corner or in the center of the drawing surface. You might find yourself guessing at the background of the format creator based on the implied origin and orientation found in the format. Some formats include provisions for the specification of the origin and orientation.

An image displayed by an application incorporating an incorrect assumption about the origin point or orientation may appear upside down or backwards, or may be shifted horizontally some fraction of the width of the drawing surface on the output device.

Sometimes the header will contain a text description field, which is a comment section consisting of ASCII data describing the name of the image, the name of the image file, the name of the person who created the image, or the software application used to create it. This field may contain 7-bit ASCII data, for portability of the header information across platforms.

Unused Space

At the end of the header may be an unused field, sometimes referred to as *padding, filler, reserved space,* or *reserved fields.* Reserved fields contain no data, are undocumented and unstructured and essentially act as placeholders. All we know about them are their sizes and positions in the header. Thus, if the format is altered at some future date to incorporate new data, the reserved space can be used to describe the format or location of this data while still maintaining backward compatibility with programs supporting older versions of the format. This is a common method used to minimize version problems—creating an initial version based on a fixed header substantially larger than necessary. New fields can then be added to reserved areas of the header in subsequent revisions of the format without altering the size of the header.

Ex.1007 / Page 95 of 1158
APPLE INC.

Often format headers are intentionally padded using this method to 128, 256, or 512 bytes. This has some implications for performance, particularly on older systems, and is designed to accommodate common read and write buffer sizes. Padding may appear after the documented fields at the end of the header, and this is sometimes an indication that the format creator had performance and caching issues in mind when the format was created.

Reserved fields are sometimes only features left over from early working versions of the format, unintentionally frozen into place when the format was released. A vendor will normally change or extend a file format only under duress, or as a rational response to market pressure typically caused by an unanticipated advance in technology. In any case, the upgrade is almost always unplanned. This usually means that a minimal amount of effort goes into shoehorning new data into old formats. Often the first element sacrificed in the process is complete backward compatibility with prior format versions.

Examples of Bitmap Headers

To give you some idea about what to expect when looking at bitmap headers, we'll take a look at three typical ones. We'll start with one of the least complex (Microsoft Windows Bitmap), and proceed to two that are more complex (Sun Raster and Kofax raster).

To do this, we'll provide a C data structure, which will provide an idea of the size and relative position of each field in the headers.

Example 1: Microsoft Windows Bitmap Version 1.x Format Header

```
//
// Header structure for the MS Windows 1.x Bitmap Format
//    a BYTE is an unsigned char
//    a WORD is an unsigned short int (16-bits)
//
typedef struct _WinHeader1x
{
//
//  Type     Name         Offset    Comment
//
    WORD     Type;        /* 00h    File Type Identifier (always 0)  */
    WORD     Width;       /* 02h    Width of Bitmap in Pixels        */
    WORD     Height;      /* 04h    Height of Bitmap in Scan-lines   */
    WORD     Width;       /* 06h    Width of Bitmap in Bytes         */
    BYTE     Planes;      /* 08h    Number of Color Planes           */
    BYTE     BitsPixel;   /* 09h    Number of Bits Per Pixel         */
} OLDWINHEAD;
```


Ex.1007 / Page 96 of 1158
APPLE INC.

As you can see from the comments, this particular header contains a file identification value, the width and height of the image, the width of a single line of the image (in bytes), the number of color planes, and the number of bits per pixel. This information is close to the bare minimum required to describe a bitmap image so it can be read and rendered in an arbitrary environment.

Note that the Windows 1.*x* header contains no information about color or image data compression. A more advanced image format will have provisions for both color information and at least a simple compression scheme. An example of a more elaborate header is that found in the Sun Raster image file format shown in the next example:

Example 2: Sun Raster Format Header

```
//
// Header structure for the Sun Raster Image File Format
//  a WORD here is an unsigned long int (32 bits)
//
typedef struct _SunRasterHead
{
//
//  Type    Name         Offset   Comment
//
    WORD    Magic;       /* 00h   Magic Number (59a66a95h)      */
    WORD    Width;       /* 04h   Width of Image in Pixels      */
    WORD    Height;      /* 08h   Height of Image in Pixels     */
    WORD    Depth;       /* 0Ch   Number of Bits Per Pixel      */
    WORD    Length;      /* 10h   Length of Image in Bytes      */
    WORD    Type;        /* 14h   File Format Encoding Type     */
    WORD    MapType;     /* 18h   Type of Color Map             */
    WORD    MapLength;   /* 1Ch   Length of Color Map in Bytes  */
} SUNRASHEAD;
```

The Sun Raster header contains information similar to the Windows 1.*x* bitmap header illustrated above. But note that it also contains fields for the type of data encoding method and the type and size of the color map or *palette* associated with the bitmap data.

Neither of the two headers mentioned above contains a text description field. One such header that does is that associated with the Kofax Image File Format shown in Example 3.

Example 3: Kofax Raster Format Header

```
//
//  Header structure for the Kofax Raster Image File Format
//    a LONG is a signed long int (32 bits)
//    a SHORT is a signed short int (16 bits)
//
```


Ex.1007 / Page 97 of 1158
APPLE INC.

```
typedef struct _KofaxHeader
{
//
//  Type     Name              Offset   Comment
//
    LONG     Magic;            /* 00h   Magic Number (68464B2Eh)     */
    SHORT    HeaderSize;       /* 04h   Header Size                  */
    SHORT    HeaderVersion;    /* 06h   Header Version Number        */
    LONG     ImageId;          /* 0Ah   Image Identification Number  */
    SHORT    Width;            /* 0Ch   Image Width in Bytes         */
    SHORT    Length;           /* 0Eh   Image Length in Scan-lines   */
    SHORT    Format;           /* 10h   Image Data Code (Encoding)   */
    CHAR     Bitsex;           /* 11h   Non-zero if Bitsex Reversed  */
    CHAR     Color;            /* 12h   Non-zero if Color Inverted   */
    SHORT    Xres;             /* 14h   Horizontal Dots Per Inch     */
    SHORT    Yres;             /* 16h   Vertical Dots Per Inch       */
    CHAR     Planes;           /* 18h   Number of Planes             */
    CHAR     BitsPerPixel;     /* 19h   Number of Bits Per Pixel     */
    SHORT    PaperSize;        /* 1Ah   Original Paper Size          */
    CHAR     Reserved1[20];    /* 1Ch   20-byte Reserved Field       */
    LONG     Dcreated;         /* 30h   Date Created                 */
    LONG     Dmodified;        /* 34h   Date Modified                */
    LONG     Daccessed;        /* 38h   Date Accessed                */
    CHAR     Reserved2[4];     /* 3Ch   4-Byte Reserved Field        */
    LONG     Ioffset;          /* 40h   Index Text Info Offset       */
    LONG     Ilength;          /* 44h   Index Text Info Length       */
    LONG     Coffset;          /* 48h   Comment Text Offset          */
    LONG     Clength;          /* 4Ch   Comment Text Length in Bytes */
    LONG     Uoffset;          /* 50h   User Data Offset             */
    LONG     Ulength;          /* 54h   User Data Length in Bytes    */
    LONG     Doffset;          /* 58h   Image Data Offset            */
    LONG     Dlength;          /* 5Ch   Image Data Length in Bytes   */
    CHAR     Reserved3[32];    /* 60h   32-byte Reserved Field       */
} KFXHEAD;
```

Note that the Kofax header is considerably larger than either the Windows bitmap or Sun raster headers. Included are fields which describe the horizontal and vertical resolution, paper size of the image subject, offset values of different types of data stored in the file, and the time and date that the image was created, last modified, and accessed.

Also note the appearance of several fields marked *reserved.* The Kofax format header is intentionally padded to 128 bytes to accommodate common read and write buffer sizes. It uses only 72 bytes of the header in the revision presented here, but is padded to 128 bytes. The Kofax format specification promises that the first 128 bytes of every Kofax image file will be the header, regardless of future revisions. Applications are thus free to ignore the reserved data, and the format is presumably designed to allow this without dire penalty. See the general discussion of reserved fields in the section called "Unused Space" earlier in this chapter.

Ex.1007 / Page 98 of 1158
APPLE INC.

Optimizing Header Reading

Header reading speed can be optimized by looking at the ways in which your application uses the data, because reading the header data can usually be performed in several ways. If only selected values in the header are needed, the application can calculate the offset of the data from some key landmark such as the start of the file. The application can then seek directly to the data value required and read the data value. The offset values appearing in the comments in the header examples above can be used as offset arguments for the seek function used.

If most of the data contained in the header is needed by the application, then it may be more convenient to read the entire header into a buffer or preallocated data structure. This can be performed quickly, taking advantage of any efficiencies provided by an integral-power-of-two block reads, as mentioned above. All of the header data will be available in memory and can be cached for use when needed. One problem, however, occurs when the byte order of the file is different from the native byte order of the system on which the file is being read. Most block read functions, for example, are not designed to supply automatic conversion of data. Another problem may arise when data structures are padded by the compiler or runtime environment for purposes of data member alignment. These problems and others are discussed in more detail in Chapter 6, *Platform Dependencies.*

Bitmap Data

The actual bitmap data usually makes up the bulk of a bitmap format file. In the following discussion, we'll assume that you've read Chapter 2, *Computer Graphics Basics,* and that you understand about pixel data and related topics like color.

In many bitmap file formats the actual bitmap data is found immediately after the end of the file header. It may be found elsewhere in the file, however, to accommodate a palette or other data structure which also may be present. If this is the case, an offset value will appear in the header or in the documentation indicating where to find the beginning of the image data in the file.

One thing you might notice while looking over the file format specifications described in Part Two of this book is the relative absence of information explaining the arrangement of the actual bitmap data in the file. To find out how the data is arranged, you usually have to figure it out from related information pertaining to the structure of the file.

Ex.1007 / Page 99 of 1158
APPLE INC.

Fortunately, the structuring of bitmap data within most files is straightforward and easily deduced. As mentioned above, bitmap data is composed of pixel values. Pixels on an output device are usually drawn in scan lines corresponding to rows spanning the width of the display surface. This fact is usually reflected in the arrangement of the data in the file. This exercise, of deducing the exact arrangement of data in the file, is sometimes helped by having some idea of the display devices the format creator had in mind.

One or more scan lines combined form a 2D grid of pixel data; thus we can think of each pixel in the bitmap as located at a specific logical coordinate. A bitmap can also be thought of as a sequence of values that logically maps bitmap data in a file to an image on the display surface of an output device. Actual bitmap data is usually the largest single part of any bitmap format file.

How Bitmap Data Is Written to Files

Before an application writes an image to a file, the image data is usually first assembled in one or more blocks of memory. These blocks can be located in the computer's main memory space or in part of an auxiliary data collection device. Exactly how the data is arranged then depends on a number of factors, including the amount of memory installed, the amount available to the application, and the specifics of the data acquisition or file write operation in use. When bitmap data is finally written to a file, however, only one of two methods of organization is normally used: scan-line data or planar data.

Scan-line data

The first, and simplest, method is the organization of pixel values into rows or scan lines, briefly mentioned above. If we consider every image to be made up of one or more scan lines, the pixel data in the file describing that image will be a series of sets of values, each set corresponding to a row of the image. Multiple rows are represented by multiple sets written from start to end in the file. This is the most common method for storing image data organized into rows.

If we know the size of each pixel in the image, and the number of pixels per row, we can calculate the offset of the start of each row in the file. For example, in an 8-bit image every pixel value is one byte long. If the image is 21 pixels wide, rows in the file are represented by sets of pixel values 21 bytes wide. In this case, the rows in the file start at offsets of 0, 21, 42, 63, etc. bytes from the start of the bitmap data.

On some machines and in some formats, however, rows of image data must be certain even-byte multiples in length. An example is the common rule

Ex.1007 / Page 100 of 1158
APPLE INC.

requiring bitmap row data to end on long-word boundaries, where a long word is four bytes long. In the example mentioned in the preceding paragraph, an image 21 pixels wide would then be stored in the file as sets of pixel values 24 bytes in length, and the rows would start at file offsets 0, 24, 48, 64. The extra three bytes per row are padding. In this particular case, three bytes of storage in the file are wasted for every row, and in fact, images that are 21 pixels wide take up the same amount of space as images 24 pixels wide. In practice, this storage inefficiency is usually (but not always) compensated for by an increase of speed gained by catering to the peculiarities of the host machine in regard to its ability to quickly manipulate two or four bytes at a time. The actual width of the image is always available to the rendering application, usually from information in the file header.

In a 24-bit image, each image pixel corresponds to a 3-byte long pixel value in the file. In the example we have been discussing, an image 21 pixels wide would require a minimum of 21 * 3 = 63 bytes of storage. If the format requires that the row starts be long-word aligned, 64 bytes would be required to hold the pixel values for each row. Occasionally, as mentioned above, 24-bit image data is stored as a series of 4-byte long pixel values, and each image row would then require 21 * 4 = 84 bytes. Storing 24-bit image data as 4-byte values has the advantage of always being long-word aligned, and again may make sense on certain machines.

In a 4-bit image, each pixel corresponds to one-half byte, and the data is usually stored two pixels per byte, although storing the data as 1-byte pixel values would make the data easier to read and, in fact, is not unheard of.

Figure 3-1 illustrates the organization of pixel data into scan lines.

Planar data

The second method of pixel value organization involves the separation of image data into two or more planes. Files in which the bitmap data is organized in this way are called planar files. We will use the term *composite image* to refer to an image with many colors (i.e., not monochrome, not gray-scale, and not one single color). Under this definition, most normal colored images that you are familiar with are composite images.

A composite image, then, can be represented by three blocks of bitmap data, each block containing just one of the component colors making up the image. Constructing each block is akin to the photographic process of making a separation—using filters to break up a color photograph into a set of component colors, usually three in number. The original photograph can be reconstructed by combining the three separations. Each block is composed of rows laid end to end, as in the simpler storage method explained above; in this case, more

Ex.1007 / Page 101 of 1158
APPLE INC.

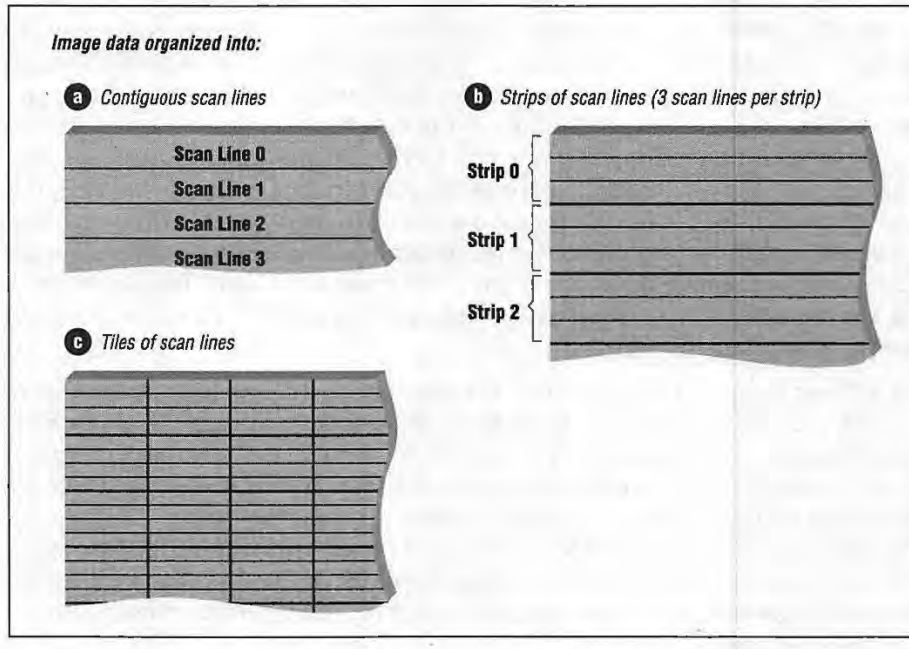


FIGURE 3-1: *Organization of pixel data into scan lines (24-bit image)*

than one block is now needed to reconstruct the image. The blocks may be stored consecutively or may be physically separated from one another in the file.

Planar format data is usually a sign that the format designer had some particular display device in mind, one that constructed composite color pixels from components routed through hardware designed to handle one color at a time. For reasons of efficiency, planar format data is usually read one plane at a time in blocks, although an application may choose to laboriously assemble composite pixels by reading data from the appropriate spot in each plane sequentially.

As an example, a 24-bit image two rows by three columns wide might be represented in RGB format as six RGB pixel values:

```
(00, 01, 02) (03, 04, 05) (06, 07, 08)
(09, 10, 11) (12, 13, 14) (15, 16, 17)
```

but be written to the file in planar format as:

```
(00) (03) (06)
(09) (12) (15)
red plane
```


Ex.1007 / Page 102 of 1158
APPLE INC.

```
(01) (04) (07)
(10) (13) (16)
green plane
(02) (05) (08)
(11) (14) (17)
blue plane
```

Notice that the exact same data is being written; it's just arranged differently. In the first case, an image consisting of six 24-bit pixels is stored as six 3-byte pixel values arranged in a single plane. In the second, planar, method, the same image is stored as 18 1-byte pixel values arranged in three planes, each plane corresponding to red, green, and blue information, respectively. Each method takes up exactly the same amount of space, 18 bytes, at least in this example.

It's pretty safe to say that most bitmap files are stored in non-planar format. Supporting planar hardware, then, usually means disassembling the pixel data and creating multiple color planes in memory, which are then presented to the planar rendering subroutine or the planar hardware.

Planar files may need to be assembled in a third buffer or, as mentioned above, laboriously set (by the routine servicing the output device) one pixel at a time.

Figure 3-2 illustrates the organization of pixel data into color planes.

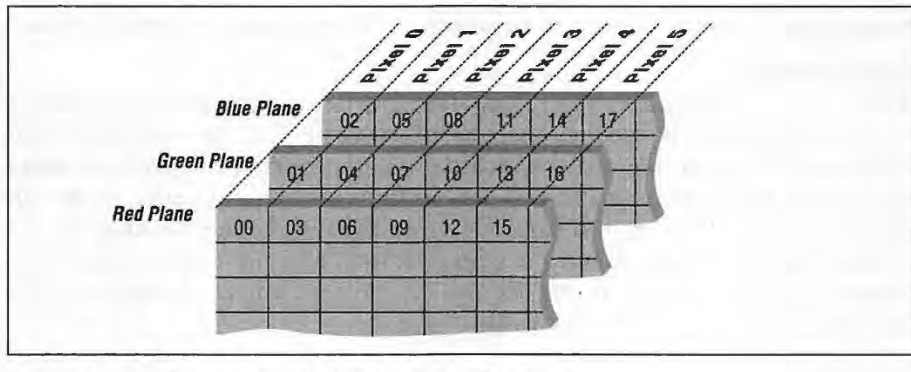


FIGURE 3-2: *Organization of pixel data into color planes*

Different Approaches to Bitmap Data Organization

Normally, we consider an image to be made up of a number of rows, each row a certain number of pixels wide. Pixel data representing the image can be stored in the file in three ways: as *contiguous data*, as *strips* or as *tiles*. Figure 3-3 illustrates these three representations.

Ex.1007 / Page 103 of 1158
APPLE INC.

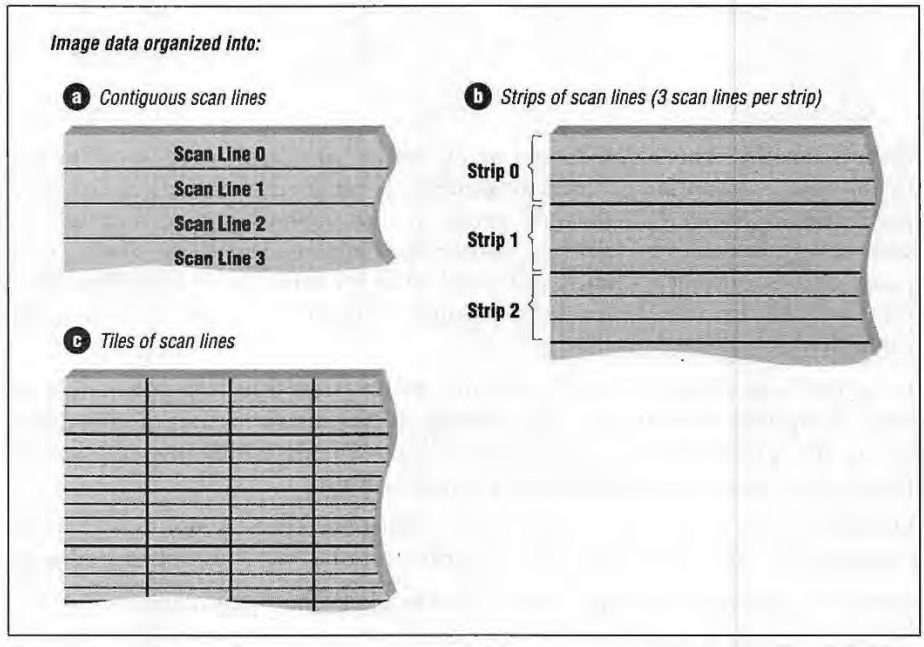


FIGURE 3-3: *Examples of bitmap data organization (contiguous scan lines, strips, and tiles)*

Contiguous data

The simplest method of row organization is where all of the image data is stored contiguously in the file, one row following the last. To retrieve the data you read the rows in file order, which delivers the rows in the order in which they were written. The data in this organizational scheme is stored in the file equivalent of a 2D array. You can index into the data in the file knowing the width of the row in pixels and the storage format and size of the pixel values. Data stored contiguously in this manner can be read quickly, in large chunks, and assembled in memory quite easily.

Strips

In the second method of file organization, images are stored in strips, which also consist of rows stored contiguously. The total image, however, is represented by more than one strip, and the individual strips may be widely separated in the file. Strips divide the image into a number of segments, which are always just as wide as the original image.

Ex.1007 / Page 104 of 1158
APPLE INC.

Strips make it easier to manage image data on machines with limited memory. An image 1024 rows long, for instance, can be stored in the file as eight strips, each strip containing 128 rows. Arranging a file into strips facilitates buffering. If this isn't obvious, consider an uncompressed 8-bit image 1024 rows long and 10,000 pixels wide, containing 10 megabytes of pixel data. Even dividing the data into eight strips of 128 rows leaves the reading application with the job of handling 1.25 megabytes of data per strip, a chore even for a machine with a lot of flat memory and a fast disk. Dividing the data into 313 strips, however, brings each strip down to a size which can be read and buffered quickly by machines capable of reading only 32K of data per file read pass.

Strips also come into play when pixel data is stored in a compressed or encoded format in the file. In this case, an application must first read the compressed data into a buffer and then decompress or decode the data into another buffer the same size as, or larger than, the first. Arranging the compression on a per-strip basis greatly eases the task of the file reader, which need handle only one strip at a time.

You'll find that strips are often evidence that a file format creator has thought about the limitations of the possible target platforms being supported and has wanted to not limit the size of images that can be handled by the format. Image file formats allowing or demanding that data be stored in strips usually provide for the storage of information in the file header such as the number of strips of data, the size of each strip, and the offset position of each strip within the file.

Tiles

A third method of bitmap data organization is tiling. Tiles are similar to strips in that each is a delineation of a rectangular area of an image. However, unlike strips, which are always the width of the image, tiles can have any width at all, from a single pixel to the entire image. Thus, in one sense, a contiguous image is actually one large tile. In practice, however, tiles are arranged so that the pixel data corresponding to each is between 4Kb and 64Kb in size and is usually of a height and width divisible by 16. These limits help increase the efficiency with which the data can be buffered and decoded.

When an image is tiled, it is generally the case that all the tiles are the same size, that the entire image is tiled, that the tiles do not overlap, and that the tiles are all encoded using the same encoding scheme. One exception is the CALS Raster Type II format, which allows image data to be composed of both encoded and unencoded image tiles. Tiles are usually left unencoded when such encoding would cause the tile data to increase in size (negative compression) or when an unreasonable amount of time would be required to encode the tile.

Ex.1007 / Page 105 of 1158
APPLE INC.

Dividing an image into tiles also allows different compression schemes to be applied to different parts of an image to achieve an optimal compression ratio. For example, one portion of an image (a very busy portion) could be divided into tiles that are compressed using JPEG, while another portion of the same image (a portion containing only one or two colors) could be stored as tiles that are run-length encoded. In this case, the tiles in the image would not all be the same uniform size; the smallest would be only a few pixels, and the largest would be hundreds or thousands of pixels on a side.

Tiing sometimes allows faster decoding and decompression of larger images than would be possible if the pixel data were organized as lines or strips. Because tiles can be individually encoded, file formats allowing the use of tiles will contain tile quantity, size, and offset information in the header specifications. Using this information, a reader that needs to display only the bottom-right corner of a very large image would have to read only the tiles for that area of the image; it would not have to read all of the image data that was stored before it.

Certain newer tile-oriented compression schemes, such as JPEG, naturally work better with file formats capable of supporting tiling. A good example of this is the incorporation of JPEG in later versions of the TIFF file format. For more information about the use of tiles, see the article on the TIFF file format in Part Two of this book.

Palette

Many bitmap file formats contain a color palette. For a discussion of palettes of different kinds, see Chapter 2.

Footer

The *footer*, sometimes called the *trailer*, is a data structure similar to a header and is often an addition to the original header, but appended to the end of a file. A footer is usually added when the file format is upgraded to accommodate new types of data and it is no longer convenient to add or change information in the header. It is mainly a result of a desire to maintain backward compatibility with previous versions of the format. An example of this is the TGA format, later revisions of which contain a footer that enables applications to identify the different versions of its format and to access special features available only in the later version of the format.

Because by definition it appears after the image data, which is usually of variable length, a footer is never found at a fixed offset from the beginning of an

Ex.1007 / Page 106 of 1158
APPLE INC.

image file unless the image data is always the same size. It is, however, usually located at a specified offset from the end of an image file. Like headers, footers are usually a fixed size. The offset value of the footer may also be present in the header information, provided there was reserved space or padding available in the header. Also like a header, a footer may contain an identification field or magic number which can be used by a rendering application to differentiate it from other data structures in the file.

Other Bitmap File Data Structures

Besides headers, footers, and palettes, bitmap files may contain additional data structures, which are usually added to aid manipulation of the image data by the rendering application.

A file format which allows more than one image in the file needs to provide some method of identifying the start of each image. Thus, an *image offset table*, sometimes called an *image file index* or *page table*, may be used to store the offset values of the start of each image from the beginning of the file.

A *scan-line table* may be provided for locating the start of each image scan line in the pixel data. This can be useful if the image data is compressed and pixel data corresponding to individual scan lines must be accessed randomly; the pixels in the image data cannot be counted until the image data is decoded. Scan-line tables contain one entry per image scan line. Variants of this idea include strip location tables (one entry per group of scan lines) and tile location tables (one entry for each rectangular subarea of the image).

Other Bitmap File Features

Several formats incorporate unique or unusual data structures in their design. These are usually to accomplish the specific purpose of the format or to create as much generality as possible.

A common file format that comes to mind under the heading of "unusual" is TIFF. TIFF contains a rudimentary header, but stores much of its data in a series of tags called Image File Directories, which are fixed in neither size nor position. They are instead like an in-memory list data structure in that they are linked by a series of file offset values. Data can be found by seeking to the next offset from the current offset. While this arrangement can lead to confusion (and indeed TIFF has many times been called a "write-only" format), it allows a programmer to construct a header-like structure that can contain any information at all, thus adding to its versatility.

Ex.1007 / Page 107 of 1158
APPLE INC.

Unusual or unique features of other formats include the storing of image data and palette information in separate files (the Dr. Halo CUT and PAL files, for example) and the storing of monochrome bitmaps as blocks of ASCII format 1's and 0's (as in the PBM format), designed perhaps with interplatform portability in mind.

Pros and Cons of Bitmap File Formats

Bitmap files are especially suited for the storage of real-world images; complex images can be rasterized in conjunction with video, scanning, and photographic equipment and stored in a bitmap format.

Advantages of bitmap files include the following:

- Bitmap files may be easily created from existing pixel data stored in an array in memory.
- Retrieving pixel data stored in a bitmap file may often be accomplished by using a set of coordinates that allows the data to be conceptualized as a grid.
- Pixel values may be modified individually or as large groups by altering a palette if present.
- Bitmap files may translate well to dot-format output devices such as CRTs and printers.

Bitmap files, however, do have drawbacks:

- They can be very large, particularly if the image contains a large number of colors. Data compression can shrink the size of pixel data, but the data must be expanded before it can be used, and this can slow down the reading and rendering process considerably. Also, the more complex a bitmap image (large number of colors and minute detail), the less efficient the compression process will be.
- They typically do not scale very well. Shrinking an image by *decimation* (throwing away pixels) can change the image in an unacceptable manner, as can expanding the image through pixel replication. Because of this, bitmap files must usually be printed at the resolution in which they were originally stored.

Ex.1007 / Page 108 of 1158
APPLE INC.

CHAPTER 4

Vector Files

In this chapter we'll be talking about vector files. Because we've already introduced bitmap files in Chapter 3, *Bitmap Files*, we'll be contrasting selected features of vector files with their bitmap counterparts.

Vector Versus Bitmap Files

A bitmap file in some sense contains an exact pixel-by-pixel mapping of an image, which can then be reconstructed by a rendering application on the display surface of an output device. Rendering applications seldom have to take into account any structural elements other than pixels, scan lines, strips, and tiles—subdivisions of the image which were made without reference to the content of the image.

Vector files contain, instead, mathematical descriptions of one or more *image elements*, which are used by the rendering application to construct a final image. Vector files are thus said to be made up of descriptions of image elements or *objects*, rather than pixel values. Although the term object has a modern meaning, you will find vector format specifications adhering to the older usage.

What Is Vector Data?

Vectors are line segments minimally defined as a starting point, a direction, and a length. They can, however, be much more complex and can include various sorts of lines, curves, and splines. Straight and curved lines can be used to define geometrical shapes, such as circles, rectangles, and polygons, which then can be used to create more complex shapes, such as spheres, cubes, and polyhedrons.

Ex.1007 / Page 109 of 1158
APPLE INC.

Vector Files Came First

Vector file formats have been around since computers were first used to display lines on an output device. CRTs, for example, were first used as computer-driven output devices in the 1950s. The first CRT displays were random scan devices similar to oscilloscopes, capable of producing images of mathematical and geometrical shapes. Vector display devices provided output sufficient for the needs of computer users for many years after their introduction, due to the limited range of tasks computers were called upon to perform.

At some point the need to store vector data arose, and portable storage media such as punch cards or paper tape were pressed into use. Prior to rendering time, an image was logically subdivided into its simplest elements. At rendering time, the image was produced and maintained by drawing each of its elements repeatedly in a specified order. At storage time, data was readily exported as a list of drawing operations, and mathematical descriptions of the image elements—their size, shape, and position on the display screen—were written to the storage device in the order in which they were displayed.

Vector Files and Device Independence

As mentioned above, vector images are collections of device-independent mathematical descriptions of graphical shapes.

More so than their bitmap counterparts, various vector formats differ primarily because each was designed for a different purpose. While the conceptual differences between the designs of formats supporting 1-bit and 24-bit bitmap data may be small, the differences between vector formats used with CAD applications and those used for general data interchange can be formidable. Thus, it is difficult to generalize about vector formats in the same way we did when discussing bitmap formats.

On the other hand, most output devices are point-addressable, providing a grid of pixels which can be addressed individually, as if the surface of the device were graph paper made of discrete elements. This means that an application can always find a way to draw vector-format image elements on the device.

Ex.1007 / Page 110 of 1158
APPLE INC.

Sources of Vector Format Files

The simplest vector formats are those used by spreadsheet applications. These normally contain numerical data meant to be displayed on a 2D grid on an output device. Some non-spreadsheet applications use spreadsheet file formats to store data that can alternately be interpreted as either bitmap or vector data.

Examples of common spreadsheet formats include those associated with the programs Lotus 1-2-3 (.WKS and .WK1), Excel (.XLS), and Quattro Pro. Although these originated on Intel-based PCs, the respective vendors now support multiple platforms. Several spreadsheet formats have been developed explicitly to support portable data interchange between different spreadsheet applications; as a result, these are now also found on multiple platforms. These include Lotus DIF (Data Interchange Format) and Microsoft SYLK (SYmbolic LinK Format).

The majority of vector formats, however, are designed for storing line drawings created by CAD applications. CAD packages are used to create mechanical, electrical, and architectural drawings, electronic layouts and schematics, maps and charts, and artistic drawings. The complexity of information needed to support the needs of a major CAD application is considerably greater than that needed to support a spreadsheet and generally requires a more complicated vector format.

CGM (Computer Graphics Metafile) is an example of a general format designed with data interchange in mind, a format that is defined in a published standard. All elements in a CGM-format file are constructed of simple objects such as lines and polygons, primitives assumed to be available in every rendering application. Very complex objects are broken down into the simplest possible shapes.

Autodesk's AutoCAD DXF (Data eXchange Format) was also designed with vector data interchange in mind but is vendor-controlled and originated as a format supporting a single application. In addition, DXF was specifically tailored for CAD information useful in the construction of mechanical, electrical, and architectural drawings. DXF therefore supports not only common vector elements such as circles and polygons, but also complex objects frequently used in CAD renderings, such as 3D objects, labels, and hatching.

How Vector Files Are Organized

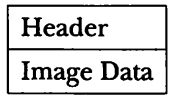
Although vector files, like bitmap files, vary considerably in design, most contain the same basic structure: a header, a data section, and an end-of-file

Ex.1007 / Page 111 of 1158
APPLE INC.

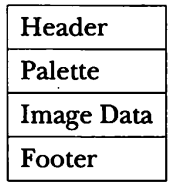
marker. Some structure is needed in the file to contain information global to the file and to correctly interpret the vector data at render time. Although most vector files place this information in a header, some rely solely on a footer to perform the same task.

Vector files on the whole are structurally simpler than most bitmap files and tend to be organized as data streams. Most of the information content of the file is found in the image data.

The basic components of a simple vector file are the following:



If a file contains no image data, only a header will be present. If additional information is required that does not fit in the header, you may find a footer appended to the file, and a palette may be included as well:



Header

The header contains information that is global to the vector file and must be read before the remaining information in the file can be interpreted. Such information can include a file format identification number, a version number, and color information.

Headers may also contain default attributes, which will apply to any vector data elements in the file lacking their own attributes. While this may afford some reduction in file size, it does so at the cost of introducing the need to cache the header information throughout the rendering operation.

Headers and footers found in vector-format files may not necessarily be a fixed size. For historical reasons mentioned above, it is not uncommon to find vector formats which use streams of variable-length records to store all data. If this is the case, then the file must be read sequentially and will normally fail to provide offset information that is necessary to allow the rendering application to subsample the image.

Ex.1007 / Page 112 of 1158
APPLE INC.

The type of information stored in the header is governed by the types of data stored in the file. Basic header information contains the height and width of the image, the position of the image on the output device, and possibly the number of layers in the image. Thus, the size of the header may vary from file to file within the same format.

Vector Data

The bulk of all but tiny files consists of vector element data that contain information on the individual objects making up the image. The size of the data used to represent each object will depend upon the complexity of the object and how much thought went into reducing the file size when the format was designed.

Following the header is usually the image data. The data is composed of *elements*, which are smaller parts that comprise the overall image. Each element either inherits information or is explicitly associated with default information that specifies its size, shape, position relative to the overall image, color, and possibly other attribute information. An example of vector data in ASCII format containing three elements (a circle, a line, and a rectangle), might appear as:

```
;CIRCLE,40,100,100,BLUE;LINE,200,50,136,227,BLACK;RECT,80,65,25,78,RED;
```

Although this example is a simple one, it illustrates the basic problem of deciphering vector data, which is the existence of multiple levels of complexity. When deciphering a vector format, you not only must find the data, but you also must understand the formatting conventions and the definitions of the individual elements. This is hardly ever the case in bitmap formats; bitmap pixel data is all pretty much the same.

In this example, elements are separated by semicolons, and each is named, followed by numerical parameters and color information. Note, however, that consistency of syntax across image elements is never guaranteed. We could have just as easily defined the format in such a way as to make blocks of unnamed numbers signify lines by default:

```
;CIRCLE,40,100,100,BLUE;200,50,136,227,BLACK;RECT,80,65,25,78,RED;
```

and the default color black if unspecified:

```
;CIRCLE,40,100,100,BLUE;200,50,136,227;RECT,80,65,25,78,RED;
```

Many formats allow abbreviations:

```
;C,40,100,100,BL;200,50,136,227;R,80,65,25,78,R;
```

Notice that the R for RECT and R for RED are distinguished by context. You will find that many formats have opted to reduce data size at the expense of

Ex.1007 / Page 113 of 1158
APPLE INC.

conceptual simplicity. You are free to consider this as evidence of flawed reasoning on the part of the format designer. The original reason for choosing ASCII was for ease of reading and parsing. Unfortunately, using ASCII may make the data too bulky. Solution: reduce the data size through implied rules and conventions and allow abbreviation (in the process making the format unreadable). The format designer would have been better off using a binary format in the first place.

After the image data is usually an end-of-section or end-of-file marker. This can be as simple as the string EOF at the end the file. For the same reasons discussed in Chapter 3, *Bitmap Files*, some vector formats also append a footer to the file. Information stored in a footer is typically not necessary for the correct interpretation of the rendering and may be incidental information such as the time and date the file was created, the name of the application used to create the file, and the number of objects contained in the image data.

Palettes and Color Information

Like bitmap files, vector files can contain palettes. (For a full discussion of palettes, see the discussion in Chapter 3.) Because the smallest objects defined in vector format files are the data elements, these are the smallest features for which color can be specified. Naturally, then, a rendering application must look up color definitions in the file palette before rendering the image. Our example above, to be correct, would thus need to include the color definitions, which take the form of a palette with associated ASCII names:

```
RED,255,0,0,
BLACK,0,0,0,
BLUE,0,0,255
;C,40,100,100,BL;200,50,136,227;R,80,65,25,78,R;
```

Some vector files allow the definition of enclosed areas, which are considered outlines of the actual vector data elements. Outlines may be drawn with variations in thickness or by using what are known as different *pen styles*, which are typically combinations of dots and dashes and which may be familiar from technical and CAD drawings. Non-color items of information necessary for the reproduction of the image by the rendering application are called *element attributes.*

Fills and color attributes

Enclosed elements may be designed to be filled with color by the rendering application. The filling is usually allowed to be colored independently from the element outline. Thus, each element may have two or more colors associated with

Ex.1007 / Page 114 of 1158
APPLE INC.

it, one for the element outline and one for the filling. *Fill colors* may be transparent, for instance, and some formats define what are called *color attributes.* In addition to being filled with solid colors, enclosed vector elements may contain hatching or shading, which are in turn called *fill attributes.* In some cases, fill and color attributes are lumped together, either conceptually in the format design, or physically in the file.

Formats that do not support fill patterns must simulate them by drawing parts of the pattern (lines, circles, dots, etc.) as separate elements. This not only introduces an uneven quality to the fill, but also dramatically increases the number of objects in the file and consequently the file size.

Gradient fills

An enclosed vector element may also be filled with more than one color. The easiest way is with what is called a *gradient fill,* which appears as a smooth transition between two colors located in different parts of the element fill area. Gradient fills are typically stored as a starting color, an ending color, and the direction and type of the fill. A rendering application is then expected to construct the filled object, usually at the highest resolution possible. CGM is an example of a format that supports horizontal, vertical, and circular gradient fills. Figure 4-1 illustrates a gradient fill.

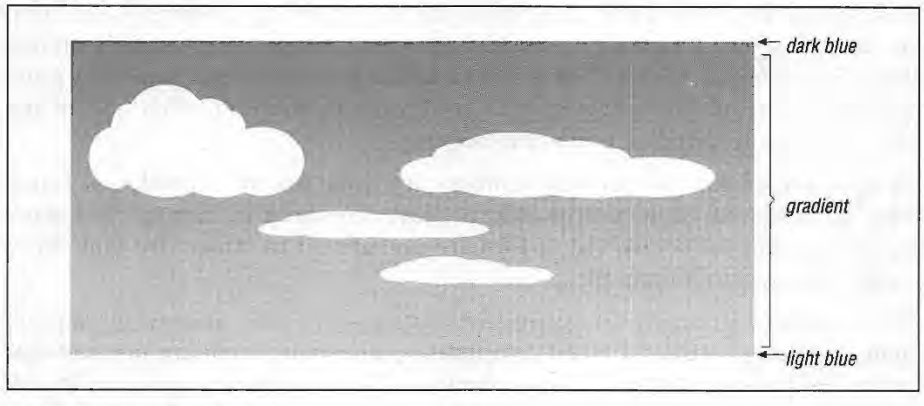


FIGURE 4-1: *Gradient fill*

Footer

A footer may contain information that can be written to the file only after all the object data is written, such as the number of objects in the image. The footer in most vector formats, however, is simply used to mark the end of the object data.

Ex.1007 / Page 115 of 1158
APPLE INC.

Vector File Size Issues

Not counting palette and attribute information, the size of a vector file is directly proportional to the number of objects it contains. Contrast this with a complex bitmap file, which stays the same size no matter how complex the image described within. The only impact complexity has on bitmap files is on the degree of compression available to the file creator.

Vector files thus can vary greatly in size. A format can store images efficiently by using some form of shorthand notation to allow the compact definition of complex elements. A vector format rich in objects might be able to represent a single complex element using a Bezier curve, for instance. Another format not supporting Bezier curves would need to represent the same curve inefficiently, perhaps using a series of lines. Each line, in this case, would be a separate element, producing a file much larger than one supporting Bezier curves directly.

A format creator was probably addressing the problem of file size when he or she decided to support the creation and naming of complex elements. Great size savings come about when elements are repeated in the image; all that needs to be stored after the original element definition is a pointer to that definition, as well as attribute and position information specific to each individual repeated element.

Size savings may also come about from the way in which a format stores information. Different formats may support identical information in widely varying ways. For example, in the CGM format a hatch pattern is represented as a single object. In the PIC and Autodesk DXF formats, however, each line in the hatch pattern is stored as a separate element.

Because vector data is stored as numbers, it can be scaled, rotated, and otherwise manipulated easily and quickly, at least compared to bitmap data. Also, because scaling is so easy, vector files are not subject to image size limitations in the same way as bitmap files.

Vector formats normally do not support data compression as most bitmap formats do. Some formats, however, support an alternate encoding method that produces smaller data files, but contains the same information. CGM, for instance, normally stores vector information in a clear-text ASCII format that is human-readable, as does the example we presented earlier in this chapter. It also allows the storage of information in a binary format, however, which results in smaller files at the cost of readability and cross-platform portability. The DXF format also has a binary analog called DXB (Data eXchange Binary) which is not only smaller, but faster to load into its parent application (AutoCAD). It is, however, not portable to other applications.

Ex.1007 / Page 116 of 1158
APPLE INC.

Scaling Vector Files

A vector element may be scaled to any size. Precision, overflow, and underflow problems may occur, however, if vector data is scaled too large or too small, "large" and "small" being relative to the intrinsic data size of the hardware and software platform supporting the rendering application. Although these problems are well known in numerical analysis, they are not generally recognized by the majority of programmers, and it pays to keep them in mind.

Another common problem occurs when apparently enclosed elements are enlarged and then rendered. Two lines meant to be joined may have endpoints slightly misaligned, and this misalignment may show up as a gap when the element is enlarged or rotated. When a rendering application attempts to display the element on an output device, fill colors or patterns may inexplicably "leak." Many applications that allow the creation of vector files have tools to prevent this, but they may not be applied automatically before the file is saved.

Text in Vector Files

Vector formats that allow the storage of text strings do so in one of two ways. The simplest approach is to store the text as a literal ASCII string along with font, position, color, and attribute information. Although the text is provided in a compact form, this scheme requires the rendering application to have a knowledge of the font to be used, which is always problematic. Because font names are for the most part vendor-controlled, it is sometimes difficult to even specify the font to be drawn. The CGM format addresses this problem through the use of an international registry of font names and associated descriptive data. Any rendering application supporting CGM must have access to this data, or it must use the font metric data supplied in the CGM file's header. Text, however, because it is stored in human-readable format, can be edited.

The second approach, and by far the most flexible, is to store the characters making up the text string as outlines constructed from a series of primitive vector data elements. Under this scheme each creator application must have access to font outline data; because it is stored like any other vector data, font outline data can be scaled at will, rotated, and otherwise manipulated. Until recently, access to outline data has been a problem, but vendors have realized the importance of support for outline fonts and are now routinely supplying this capability at the operating system level.

Because the graphics industry at large and the font industry have grown up in parallel, and only lately have begun to merge, there are naturally some

Ex.1007 / Page 117 of 1158
APPLE INC.

incompatibilities between data storage models. Most fonts, for instance, are stored as a series of splines joined end-to-end, and a particular spline type may not be supported by the file format in use. In this case, the creator application may choose to convert the splines to arcs or lines and store these instead. This may or may not have an effect on the appearance of the text.

Creator applications may even incorporate vector or stroke fonts, which are usually primitive sets of character outlines with an angular or mechanical look, designed to be drawn with a minimum of fuss. Although many vendors have chosen to make their own, one widely available source is the Hershey fonts. Hershey data is available commercially, but is no longer considered adequate for general use.

The use of vector, stroke, or outline fonts usually increases the size of a file dramatically, but this may be offset by an increase in visual quality in the case of spline-based outline fonts. Although there are a number of older font formats still in use, spline-based outline font data in the TrueType and Adobe Type 1 formats is easily available on all the major platforms. There is seldom any need to use stroke fonts now. Unfortunately, the reconstruction of characters from spline outline data is no trivial task, and the higher quality afforded by the ready availability of TrueType and Adobe Type 1 fonts comes at a price in rendering time and program development costs.

Pros and Cons of Vector Files

Advantages of vector files include the following:

- Vector files are useful for storing images composed of line-based elements such as lines and polygons, or those that can be decomposed into simple geometrical objects, such as text. More sophisticated formats can also store 3D objects such as polyhedrons and wire-frame models.
- Vector data can be easily scaled and otherwise manipulated to accommodate the resolution of a spectrum of output devices.
- Many vector files containing only ASCII-format data can be modified with simple text editing tools. Individual elements may be added, removed, or changed without affecting other objects in the image.
- It is usually easy to render vector data and save it to a bitmap format file, or, alternately, to convert the data to another vector format, with good results.

Some drawbacks of vector files include the following:

- Vector files cannot easily be used to store extremely complex images, such as some photographs, where color information is paramount and may vary on a pixel-by-pixel basis.

Ex.1007 / Page 118 of 1158
APPLE INC.

- The appearance of vector images can vary considerably depending upon the application interpreting the image. Factors include the rendering application's compatibility with the creator application and the sophistication of its toolkit of geometric primitives and drawing operations.
- Vector data also displays best on vectored output devices such as plotters and random scan displays. High-resolution raster displays are needed to display vector graphics as effectively.
- Reconstruction of vector data may take considerably longer than that contained in a bitmap file of equivalent complexity, because each image element must be drawn individually and in sequence.

Ex.1007 / Page 119 of 1158
APPLE INC.

CHAPTER 5

Metafiles

Metafiles can contain both bitmap and vector data.

When the term *metafile* first appeared, it was used in discussions of device- and machine-independent interchange formats. In the mid-1970s, the National Center for Atmospheric Research (NCAR), along with several other research institutions, reportedly used a format called metacode, which was device- and platform-independent to a certain degree. What is known for certain is that in 1979, the SIGGRAPH Graphics Standards and Planning Committee used the term, referring to a part of their published standards recommendations. These early attempts at defining device- and platform-independent formats mainly concerned themselves with vector data. Although work has continued along this line, we will refer to formats that can accommodate both bitmap and vector data as metafiles, because for all practical purposes the interchange formats in widespread use in the marketplace handle both types of data.

Although metafile formats may be used to store only bitmap or only vector information, it is more likely that they will contain both types of data. From a programmer's point of view, bitmap and vector data are two very different problems. Because of this, supporting both bitmap and vector data types adds to the complexity of a format. Thus, programmers find themselves avoiding the use of metafile formats unless the added complexity is warranted—either because they need to support multiple data types or for external reasons.

The simplest metafiles resemble vector format files. Historically, limitations of vector formats were exceeded when the data that needed to be stored became complex and diverse. Vector formats were extended conceptually, allowing the definition of vector data elements in terms of a language or grammar, and also by allowing the storage of bitmap data. In a certain sense, the resulting formats went beyond the capabilities of both bitmap and vector formats—hence the term *metafile.*

Ex.1007 / Page 120 of 1158
APPLE INC.

Platform Independence?

Metafiles are widely used to transport bitmap or vector data between hardware platforms. The character-oriented nature of ASCII metafiles, in particular, eliminates problems due to byte ordering. It also eliminates problems encountered when transferring files across networks where the eighth bit of each byte is stripped off, which can leave binary files damaged beyond repair. Also, because a metafile supports both bitmap and vector data, an application designer can kill two birds with one stone by providing support for one metafile rather than two separate bitmap and vector formats.

Metafiles are also used to transfer image data between software platforms. A creator application, for instance, can save an image in both bitmap and vector form in a metafile. This file may then be read by any bitmap-capable or vector-capable application supporting the particular metafile format. Many desktop publishing programs, for instance, can manipulate and print vector data, but are unable to display that same data on the screen. To accommodate this limitation, a bitmap representation of the image is often included along with the vector data in a metafile. The application can read the bitmap representation of the image from the metafile, which serves as a reduced-quality visual representation of the image that will eventually appear on the printed page. When the page is actually printed, however, the vector data from the metafile is used to produce the image on the printer. Display PostScript files are an example of this type of arrangement.

How Metafiles Are Organized

Metafiles vary so widely in format that it is pointless to attempt to construct a hierarchical explanation of their general construction. Most metafiles contain some sort of header, followed by one or more sections of image data. Some metafiles contain nothing but bitmap data, and still others contain no data at all, opting instead for cryptic drawing instructions, or numerical data similar to that found in vector files.

Pros and Cons of Metafiles

Because metafiles are in a sense a combination of bitmap and vector formats, many of the pros and cons associated with these formats also apply to metafiles. Your decision to choose one particular metafile format over another will thus depend on what kind of data (bitmap or vector) makes up the bulk of the file, and on the strengths and weaknesses of that particular type of data. With that said, we can safely generalize as follows:

Ex.1007 / Page 121 of 1158
APPLE INC.

- Although many metafile formats are binary, some are character-oriented (ASCII), and these are usually very portable between computer systems.
- Metafiles containing mixtures of vector and bitmap data can in some cases be smaller than fully-rendered bitmap versions of the same image.
- Because they can contain high-redundancy ASCII-encoded data, metafiles generally compress well for file transfer.
- Most metafiles are very complex, because they are usually written by one application for another application. Although their ASCII nature means that theoretically they may be modified with a text editor, modification of a metafile by hand generally requires a skilled eye and special knowledge.

Ex.1007 / Page 122 of 1158
APPLE INC.

CHAPTER 6

Platform Dependencies

One of our criteria for choosing the formats discussed in this book was whether they are used for data exchange (both between applications and across platforms). This analysis necessarily ruled out formats incorporating hardware-specific instructions (for example, printer files). Although the formats we discuss here do not raise many hardware issues, several machine dependency issues do come up with some regularity. Two of these issues have some practical implications beyond being simply sources of annoyance. This chapter describes those issues. It also touches on differences between filenames among different platforms. These are significant only because filenames may offer clues about the origins of files you may receive and need to convert.

Byte Order

We generally think of information in memory or on disk as being organized into a series of individual bytes of data. The data is read sequentially in the order in which the bytes are stored. This type of data is called byte-oriented data and is typically used to store character strings and data created by 8-bit CPUs.

Few computers look at the universe through an 8-bit window, however. For reasons of efficiency, 16-, 32-, and 64-bit CPUs prefer to work with bytes organized into 16-, 32-, and 64-bit cells, which are called words, doublewords, and quadwords, respectively. The order of the bytes within word-, doubleword-, and quadword-oriented data is not always the same; it varies depending upon the CPU that created it. (Note, however, that CPUs do exist in which byte ordering can be changed.)

Ex.1007 / Page 123 of 1158
APPLE INC.

Byte-oriented data has no particular order and is therefore read the same on all systems. Word-oriented data does present a potential problem—probably the most common portability problem you will encounter when moving files between platforms. The problem arises when binary data is written to a file on a machine with one byte order and is then read on a machine assuming a different byte order. Obviously, the data will be read incorrectly.

It is the order of the bytes within each word and doubleword of data that determine the "endianness" of the data. The two main categories of byte-ordering schemes are called *big-endian* and *little-endian.** Big-endian machines store the most significant byte (MSB) at the lowest address in a word, usually referred to as byte 0. Big-endian machines include those based on the Motorola MC68000A series of CPUs (the 68000, 68020, 68030, 68040, and so on), including the Commodore Amiga, the Apple Macintosh, and some UNIX machines.

Little-endian machines store the least significant byte (LSB) at the lowest address in a word. The two-byte word value, 1234h, written to a file in little-endian format, would be read as 3412h on a big-endian system. This occurs because the big-endian system assumes that the MSB, in this case the value 12h, is at the lowest address within the byte. The little-endian system, however, places the MSB at the highest address in the byte. When read, the position of the bytes in the word are effectively flipped in the file-reading process by the big-endian machine. Little-endian machines include those based on the Intel iAPX86 series of CPUs (the 8088, 80286, 80386, 80486, and so forth), including the IBM PC and clones.

A third term, *middle-endian*, has been coined to refer to all byte-ordering schemes that are neither big-endian nor little-endian. Such middle-endian ordering schemes include the 3-4-1-2, 2-1-4-3, 2-3-0-1, and 1-0-3-2 packed-decimal formats. The Digital Equipment Corporation PDP-11 is an example of a middle-endian machine. The PDP-11 has a DWORD byte-ordering scheme of 2-3-0-1.

The I/O routines in the C standard library always read word data in the native byte order of the machine hosting the application. This means that functions such as *fread*() and *fwrite*() have no knowledge of byte order and cannot provide needed conversions. Most C libraries, however, contain a function named *swab*(), which is used to swap the bytes in an array of bytes. While *swab*() can be used to convert words of data from one byte order to another, doing so can be

* The terms big-endian and little-endian were originally found in Jonathan Swift's book, *Gulliver's Travels*, as satirical descriptions of politicians who disputed whether eggs should be broken at their big end or their little end. This term was first applied to computer architecture by Danny Cohen. (See "For Further Information" below.)

Ex.1007 / Page 124 of 1158
APPLE INC.

inefficient, due to the necessity of making multiple calls for words greater than two bytes in size.

Programmers working with bitmap files need to be concerned about byte order, because many popular formats such as Macintosh Paint (MacPaint), Interchange File Format (IFF or AmigaPaint), and SunRaster image files are always read and written in big-endian byte order. The TIFF file format is unique, however, in that any TIFF file can be written in either format, and any TIFF reader must be able to read either byte order correctly regardless of the system on which the code is executing.

File Size and Memory Limitations

The second most common problem, after byte-ordering differences, is the handling of large files. Some systems have limited memory, as is the case with MS-DOS-based, early Macintosh, and other desktop machines. Two problems can arise as a consequence of this limitation. The first is that buffer memory available may not be adequate to handle chunks of data deemed reasonable on a larger machine. Some formats are designed with the limitations of small machines in mind. A prudent thing to do is to avoid forcing the rendering application to buffer more than 32K of data at a time.

Many problems come from hardware limitations, or from interactions between hardware and software. The most notorious example of this is the 64K memory segmentation (known as "chunking" in some environments) on MS-DOS-based Intel machines. Other problems come from software or operating system peculiarities. An example of this is the early Macintosh. This system ostensibly presented a large, flat address space to the programmer. Many programs written with the early Macintosh in mind, however, only handle memory in 32K chunks. Although the trend has been in recent years to restrict programs to 32-bit platforms or higher when there is a choice, data files produced on large machines will continue to be used on smaller machines.

As the speeds of modern machines outstrip some practical limits, such as the speed of main memory and hard disk speeds, the amount of local CPU cache memory becomes more important to system performance. Fast cache memory is relatively expensive compared to main system memory, which tends to encourage system designers to limit the size of cache memory. Nevertheless, it's not uncommon to see a megabyte or more of cache memory in some modern systems. Files can be optimized to make use of this cache, though it's not clear at this time how to make this optimization work across hardware platforms.

Ex.1007 / Page 125 of 1158
APPLE INC.

The second problem is absolute file size. As suggested above, an uncompressed 24-bit bitmap file 1024 pixels square will be a minimum of 3,145,728 bytes in size. While this much memory may be available on a workstation, it may not be on a smaller machine or on a larger machine in a multitasking situation. In this case, the rendering application will not be able to assemble the data in memory. If any alteration to the image must be done, extraordinary measures may need to be taken by the application prior to rendering. Thus, it is prudent to take advantage of the file "chunking" features available in many formats. Although it may take more programming effort to accommodate smaller machines, the effort also guarantees portability to future platforms.

Floating-Point Formats

Vector file formats occasionally store key points in floating-point format, and a number of different floating-point formats are in common use. Most floating-point data, however, is stored in a portable manner. The least common denominator approach is to store floating-point numbers as ASCII data, as a series of point pairs:

```
1234.56 2345.678 987.65 8765.43
```

The main problems you will encounter with floating-point data stored in ASCII format are with formatting conventions—how the numbers are delimited (comma, whitespace, etc.), and how many digits of precision need to be handled by your parsing routines. Library routines are readily available to handle conversion from ASCII to native binary floating-point formats.

Floating-point numbers stored in binary format present different problems. There are a number of floating-point binary formats in common use, including IEEE, Digital Equipment Corporation, and Microsoft Basic. Library routines are available for these conversions, but it may take some searching to find the correct one for your application. Sometimes, however, the hardest part of the job is identifying the formats you are trying to convert from and to.

Bit Order

Bit order refers to the direction in which bits are represented in a byte of memory. Just as words of data may be written with the most significant byte or the least significant byte first (in the lowest address of the word), so too can bits within a byte be written with the most significant bit or the least significant bit first (in the lowest position of the byte).

Ex.1007 / Page 126 of 1158
APPLE INC.

The bit order we see most commonly is the one in which the zeroth, or least significant bit, is the first bit read from the byte. This is referred to as *up bit ordering* or normal bit direction. When the seventh, or most significant, bit is the first one stored in a byte, we call this *down bit ordering*, or reverse bit direction.

The terms big-endian and little-endian are sometimes erroneously applied to bit order. These terms were specifically adopted as descriptions of differing byte orders only and are not used to differentiate bit orders (see the section called "Byte Order" earlier in this chapter).

Normal bit direction, least significant bit to most significant bit, is often used in transmitting data between devices, such as FAX machines and printers, and for storing unencoded bitmapped data. Reverse bit direction, most significant bit to least significant bit, is used to communicate data to display devices and in many data compression encoding methods. It is therefore possible for a bitmap image file to contain data stored in either or both bit directions if both encoded and unencoded data is stored in the file (as can occur in the TIFF and CALS Raster image file formats).

Problems that occur in reading or decoding data stored with a bit order that is the reverse of the expected bit order are called *bit sex* problems. When the bit order of a byte must be changed, we commonly refer to this as reversing the bit sex.

Color sex problems result when the value of the bits in a byte are the inverse of what we expect them to be. Inverting a bit (also called flipping or toggling a bit) is to change a bit to its opposite state. A 1 bit becomes a 0 and a 0 bit becomes a 1. If you have a black-on-white image and you invert all the bits in the image data, you will have a white-on-black image. In this regard, inverting the bits in a byte of image data is normally referred to as inverting the color sex.

It is important to realize that inverting and reversing the bits in a byte are not the same operation and rarely produce the same results. Note the different resulting values when the bits in a byte are inverted and reversed:

Original Bit Pattern: 10100001
Inverted Bit Pattern: 01011110

Original Bit Pattern: 10100001
Reversed Bit Pattern: 10000101

Ex.1007 / Page 127 of 1158
APPLE INC.

There are, however, cases when the inverting or reversing of a bit pattern will yield the same value:

Original Bit Pattern: 01010101
Inverted Bit Pattern: 10101010

Original Bit Pattern: 01010101
Reversed Bit Pattern: 10101010

Occasionally, it is necessary to reverse the order of bits within a byte of data. This most often occurs when a particular hardware device, such as a printer, requires that the bits in a byte be sent in the reverse order in which they are stored in the computer's memory. Because it is not possible for most computers to directly read a byte in a reversed bit order, the byte value must be read and its bits rewritten to memory in reverse order.

Reversing the order of bits within a byte, or changing the bit sex, may be accomplished by calculation or by table look-up. A function to reverse the bits within a byte is shown below:

```
//
// Reverse the order of bits within a byte.
// Returns: The reversed byte value.
//
BYTE ReverseBits(BYTE b)
{
  BYTE c;
  c  = ((b >>  1) & 0x55) | ((b <<  1) & 0xaa);
  c |= ((b >>  2) & 0x33) | ((b <<  2) & 0xcc);
  c |= ((b >>  4) & 0x0f) | ((b <<  4) & 0xf0);
    return(c);
}
```

If an application requires more speed in the bit-reversal process, the above function can be replaced with the REVERSEBITS macro and look-up table below. Although the macro and look-up table is faster in performing bit reversal than the function, the macro lacks the prototype checking that ensures that every value passed to the function *ReverseBits*() is an 8-bit unsigned value. An INVERTBITS macro is also included for color sex inversion.

```
#define INVERTBITS(b)    (~(b))
#define REVERSEBITS(b)   (BitReverseTable[b])

static BYTE BitReverseTable[256] =
{
0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
```


Ex.1007 / Page 128 of 1158
APPLE INC.

```
0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff
};
```

Filenames

Whether you are writing a file or reading one written by another user, you need to be aware of the differences among filenames on various platforms.

Filename Structure

By number of installed machines, the three most popular platforms at the time of this writing are MS-DOS, Macintosh, and UNIX, roughly in the ratio of 100:10:5. All three support the *name.extension* filenaming convention (although this is mostly true of the MS-DOS and UNIX systems). Applications occasionally use the extension portion of the filename for file type identification.

Other systems with a large installed user base (such as OS/2, Amiga, Atari, and VMS) have roughly similar naming conventions. VMS, for instance, uses as a default the format:

name1.name2;version

where *version* is an integer denoting the revision number of the file. In any case, files are likely to come from anywhere, and examination of the extension portion of a filename, if present, may help you to identify the format.

Ex.1007 / Page 129 of 1158
APPLE INC.

Filename Length

UNIX and Macintosh users are accustomed to long filenames:

ThisIsAMacFilename
This is also a Mac Filename
This.Is.A.Unix.Filename

The MS-DOS, Windows NT, and OS/2 FAT filesystems, on the other hand, limit filenames to the 8.3 format (eight characters per filename, three per extension):

msdosnam.ext

For interplatform portability, we suggest that you consider using the 8.3 convention. Be aware, if you are using MS-DOS, that you may get filename duplication errors when you convert multiple files from other platforms. Depending on the application doing the filename conversion, the following files from a typical UNIX installation:

thisis.file.number.1
thisis.file.number.2

are both converted to the following filename under MS-DOS, and the second file will overwrite the first:

thisis.fil

Case Sensitivity

Users on Macintosh and UNIX systems are accustomed to the fact that filenames are case-sensitive, and that filenames can contain mixed uppercase and lowercase. Filenames on MS-DOS systems, however, are case-insensitive, and the filesystem effectively converts all names to uppercase before manipulating them. Thus:

AMacFile.Ext
AUnixFile.Ext

become:

AMACFILE.EXT
AUNIXFIL.EXT

under MS-DOS and other similar filesystems.

Ex.1007 / Page 130 of 1158
APPLE INC.

Similarly:

Afile.Ext
AFile.Ext

are both converted to:

AFILE.EXT

For Further Information

For an excellent description of the war between the byte orders, see Danny Cohen's paper, cited below. A good description of the original of the "endian" terms may be found in Eric Raymond's monumental work, also cited below. Both publications are also widely available via the Internet.

Cohen, Danny, "On Holy Wars and a Plea for Peace," *IEEE Computer Magazine*, Volume 14, October 1981, pp. 48–54.

Raymond, Eric, *The New Hacker's Dictionary*, MIT Press, Cambridge, MA, 1991.

Ex.1007 / Page 131 of 1158
APPLE INC.

CHAPTER 7

Format Conversion

Programmers of all kinds always ask for information on how to convert between file formats, and graphics programmers are no exception. You must realize, however, that not every format can be converted to every other format, and this is doubly so if you wish to preserve image quality.

Is It Really Possible?

The biggest problems occur when you attempt to convert between files of different basic format types—bitmap to vector, for instance. Successful conversion between basic format types is not always possible due to the great differences in the ways data is stored.

File conversion is a thankless task for any number of reasons. No vendor, for instance, feels obligated to disclose revisions to proprietary formats, or even to publish accurate and timely specifications for the ones already in common use. Many formats also have obscure variants which may be difficult to track down. At any moment, too, a revision to a major application may appear, containing a bug which makes it produce incorrect files. These files will ever after need to be supported. For all these reasons, think long and hard about any decision you make about whether to include file conversion features in your application. Remember, too, that a format that is reasonably well designed for a particular intended application cannot necessarily be converted for use with another application. Interchange between devices and applications is only as good as the software components (readers and writers) which generate and interpret the format, whether it is CGM, PostScript, or any other format.

Ex.1007 / Page 132 of 1158
APPLE INC.

Don't Do It If You Don't Need to . . .

If you do need to convert image files between formats, consider using one or more of the software packages written especially for file format conversion. There are many very good tools for format conversion, which are freely available and distributed in source or binary form. We've included many such packages on the CD-ROM. There are also some very good commercial packages available. See the section called "Converting Formats" in Chapter 1, *Introduction*, for some recommendations.

You should consider writing your own conversion program only if you have very specific conversion needs not satisfied by the many publicly available and commercial applications—for example, to accommodate a proprietary format. If you do decide to write your own converter, you will need to know a bit about what to expect when converting from one format type to another.

. . . But If You Do

In what follows we will call the file to be converted (which already exists in one format) the *original*, and the file produced in the second format (after the conversion operation) the *converted* file. A *conversion application* acts upon an *original file* to produce a *converted file.*

If you remember our terminology, an application renders a graphics file to an output device or file. By extension, then, we will say that a conversion application *renders* an original file to a converted file. We will also use the term *translation* as a synonym for *conversion.*

Bitmap to Bitmap

Converting one bitmap format to another normally yields the best results of all the possible conversions between format types. All bitmap images consist of pixels, and ultimately all bitmap data is converted one pixel value at a time. Bitmap headers and the data contained in them can vary considerably, but the data contained in them can be added or discarded at the discretion of the conversion software to make the best conversion possible.

Usually, some rearrangement of the color data is necessary. This might take the form of separation of pixel data into color plane data, or the addition or removal of a palette. It might even entail conversion from one color model to another.

Ex.1007 / Page 133 of 1158
APPLE INC.

Unsuccessful bitmap-to-bitmap conversions occur most often when translating a file written in a format supporting deep pixel data to one written in a lesser-color format, for example, one supporting only palette color. This can occur when you are converting between a format supporting 24-bit data and one supporting only 8-bit data. The palette color format may not support the storage of enough data necessary to produce an accurate rendering of the image. Usually, some image-processing operations (*quantization* or *dithering*, most likely) are needed to increase the apparent quality of the converted image. Operations of this sort will necessarily make the converted image appear different from the original, and thus technically the image will not have been preserved in the conversion process.

The other main problem comes about when the converted image must be made smaller or larger than the original. If the converted image is smaller than the original, information in the original image must be thrown away. Although various image-processing strategies can be used to improve image quality, some of the information is nonetheless removed when the original file is shrunk. If the converted image is larger than the original, however, information must be created to fill in the spaces that appear between formerly adjacent pixels when the original image is enlarged. There is no completely satisfactory way to do this, and the processes currently used typically give the enlarged image a block-pixel look.

An example of a bitmap-to-bitmap conversion that is almost always successful is PCX to Microsoft Windows Bitmap.

Vector to Vector

Conversion between vector formats—for example, from AutoCAD DXF to Lotus DIF—is possible and sometimes quite easy. Two serious problems can occur, though. The first comes about due to differences in the number and type of objects available in different vector formats. Some formats, for instance, provide support for only a few simple image elements, such as circles and rectangles. Richer formats may also provide support for many additional complex elements, such as pattern fills, drop shadowing, text fonts, b-splines, and Bezier curves. Attempting to convert a file written in a complex format rich in elements to a simpler format will result in an approximation of the original image.

The second problem comes from the fact that that each vector format has its own interpretation of measurements and the appearance of image elements and primitives. Rarely do two formats agree exactly on the placement and

Ex.1007 / Page 134 of 1158
APPLE INC.

appearance of even simple image elements. Common problems are those related to line joint styles and end styles, and to centerline and centerpoint locations. For example, the Macintosh PICT format assumes that lines are drawn with a pen point that is below and to the right of the specified coordinates, while most other formats center their pens directly on the coordinates. Another example is the GEM VDI format, which assumes that a line should be drawn so that it extends one-half of the width of the line past the end coordinate. Lines in other formats often stop exactly at the end of the coordinate pixel.

It is quite difficult to generalize further than this. If you write a vector-to-vector format converter, you must be aware of the peculiarities of each vector format and the problems of conversion between one specific format and the other.

Metafile to Metafile

Because metafiles can contain both bitmap and vector image data in the same file, problems inherent in bitmap and vector conversion apply to metafiles as well. Generally, the bitmap part of the metafile will convert with success, but the accuracy of the conversion of the vector part will depend on the match to the format to which you are converting. An example of a metafile-to-metafile conversion is Microsoft Windows Metafile (WMF) to CGM.

Vector and Metafile to Bitmap

Converting vector and metafile format files to bitmap format files is generally quite easy. A vector image can be turned into a bitmap simply by dividing it up into component pixels and then writing those pixels to an array in memory in the memory equivalent of a contrasting color. The array can then be saved in a bitmap format file. This process is familiar to users of paint programs, where a mouse or stylus is used to draw geometrical shapes which appear as series of lines on the screen. When the data is written out to disk, however, it is stored in a bitmap file as a series of colored pixels rather than as mathematical data describing the position of the lines making up the image. The ultimate quality of the resulting bitmap image will depend on the resolution of the bitmap being rendered to and the complexity (color, pixel depth, and image features) of the original vector image.

The most common problem occurring with this type of conversion is *aliasing*, sometimes known as the *jaggies*. This is where arcs and diagonal lines take on a staircase appearance, partly due to the relatively low resolution of the output bitmap compared to that necessary to adequately support rendering of the output image.

Ex.1007 / Page 135 of 1158
APPLE INC.

The conversion of ASCII metafile data to binary bitmap data is usually the most complicated and time-consuming part of metafile conversion. As mentioned above in the section discussing the three basic formats, many metafile formats also contain a bitmap image. If conversion from vector to bitmap data achieves poor results, then converting the bitmap data to the desired format may not only result in a better job, but may also be a much quicker process.

A metafile-to-bitmap conversion that is almost always successful is Microsoft Windows Metafile to Microsoft Windows Bitmap.

Bitmap and Metafile to Vector

Converting bitmap and metafile format files to vector format files is usually the hardest of all to perform, and rarely does it achieve any kind of usable results. This is due to the fact that complex image processing algorithms and heuristics are necessary to find all the lines and edges in bitmap images. Once the outlines are found, they must be recognized and converted to their vector element equivalents, and each step is prone to error. Simple bitmap images may be approximated as vector images, usually as black-and-white line drawings, but more complex photographic-type images are nearly impossible to reproduce accurately. Nevertheless, commercial applications exist to provide various types of *edge detection* and *vectorization.* Edge detection remains an active area of research. Two examples available for Microsoft Windows are Corel Trace and Adobe Streamline.

Another problem inherent in the conversion of bitmap format files to vector is that of color. Although most bitmap files incorporate many colors, vector formats seldom provide support for more than a few. The conversion of an original bitmap file to a vector file can result in a loss of color in the converted image.

Metafiles also have the same problems associated with the conversion of their bitmap components, although many metafile formats are capable of handling the colors found in the original raster image data. Close vector reproductions of bitmap images are not usually possible unless the bitmap data is very simple.

Bitmap and Vector to Metafile

Converting bitmap format files to metafiles can be quite accurate because most metafile format files can contain a bitmap image as well. Vector format source files have a more limited range of metafile target formats to which they can successfully convert. Problems encountered in this type of conversion are the same as those occurring in bitmap-to-vector conversions.

Ex.1007 / Page 136 of 1158
APPLE INC.

A common process conversion of this type is the conversion of binary bitmap or vector image files to an ASCII metafile format such as PostScript. Although PostScript is covered only briefly in this book, it is widely used for file interchange, particularly on the Macintosh platform. Such conversions lend portability to image data designed to be moved between machines or which may be directed to a number of different output devices.

Other Format Conversion Considerations

There are other problems that occur when converting from one file format to another. One of the most vexing comes up when converting to a destination format that supports fewer colors than are contained in the original image.

Also, the number of colors in an image may not be a problem, but the specific colors contained in the original image may be. For example, consider the conversion of a 256-color image from one format to another. Both formats support images with up to 256 different colors in the bitmap, so the number of colors is not a problem. What is a problem, however, is that the source format chooses the colors that can go in the palette from a field of 16 million (24-bit palette), while the target format can store colors only from a range of 65,535 (16-bit palette). It is quite likely that the source image will contain colors not defined in the palette of the target image. The application doing the conversion will have to rely on some color aliasing scheme, which usually fails to provide satisfactory results.

Ex.1007 / Page 137 of 1158
APPLE INC.

CHAPTER 8

Working With Graphics Files

This chapter provides a good deal of somewhat loosely connected information about graphics files, file formats, and file format specifications. We discuss the issues you'll confront when you attempt to read and write graphics files (including examples of code you can use in your own programs). We also describe the use of test files, the corruption and encryption of graphics files, the potential for virus infection in these files, and the issues involved in developing your own file format and in writing the specification for that format, including ways of copyrighting and otherwise protecting your files and file formats.

Reading Graphics Data

A graphics format file reader is responsible for opening and reading a file, determining its validity, and interpreting the information contained within it. A reader may take, as its input, source image data either from a file or from the data stream of an input device, such as a scanner or frame buffer.

There are two types of reader designs in common use. The first is the *filter*. A filter reads a data source one character at a time and collects that data for as long as it is available from the input device or file. The second type is the *scanner* (also called a *parser*). Scanners are able to randomly access across the entire data source. Unlike filters, which cannot back up or skip ahead to read information, scanners can read and reread data anywhere within the file. The main difference between filters and scanners is the amount of memory they use. Although filters are limited in the manner in which they read data, they require only a small amount of memory in which to perform their function. Scanners, on the other hand, are not limited in how they read data, but as a tradeoff may require a large amount of memory or disk space in which to store data.

Ex.1007 / Page 138 of 1158
APPLE INC.

Because most image files are quite large, make sure that your readers are highly optimized to read file information as quickly as possible. Graphics and imaging applications are often harshly judged by users based on the amount of time it takes them to read and display an image file. One curiosity of user interface lore states that an application that renders an image on an output device one scan line at a time will be perceived as slower than an application that waits to paint the screen until the entire image has been assembled, even though in reality both applications may take the same amount of time.

Binary Versus ASCII Readers

Binary image readers must read binary data written in 1-, 2-, and 4-byte word sizes and in one of several different byte-ordering schemes. Bitmap data should be read in large chunks and buffered in memory for faster performance, as opposed to reading one pixel or scan line at a time.

ASCII format image readers require highly optimized string reader and parser functions capable of quickly finding and extracting pertinent information from a string of characters and converting ASCII strings into numerical values.

Reading a Graphics File Header

The type of code you use to implement a reader will vary greatly, depending upon how data is stored in a graphics file. For example, PCX files contain only little-endian binary data; Encapsulated PostScript files contain both binary and ASCII data; TIFF files contain binary data that may be stored in either the big- or little-endian byte order; and AutoCAD DXF files contain only ASCII data.

Many graphics files that contain only ASCII data may be parsed one character at a time. Usually, a loop and a series of rather elaborate nested case statements are used to read through the file and identify the various tokens of keywords and values. The design and implementation of such a text parser is not fraught with too many perils.

Gotchas

Where you can find some real gotchas is in working with graphics files containing binary data, such as the contents of most bitmap files. A few words of advice are in order, so that when you begin to write your own graphics file readers and writers you don't run into too many problems (and inadvertently damage your fists with your keyboard!).

Ex.1007 / Page 139 of 1158
APPLE INC.

When you read most bitmap files, you'll find that the header is the first chunk of data stored in the file. Headers store attributes of the graphics data that may change from file to file, such as the height and width of the image and the number of colors it contains. If a format always stored images of the same size, type, and number of colors, a header wouldn't be necessary. The values for that format would simply be hard-coded into the reader.

As it is, most bitmap file formats have headers, and your reader must know the internal structure of the header of each format it is to read. A program that reads a single bitmap format may be able to get away with seeking to a known offset location and reading only a few fields of data. However, more sophisticated formats containing many fields of information require that you read the entire header.

Using the C language, you might be tempted to read in a header from a file using the following code:

```
typedef struct _Header
{
    DWORD FileId;
    BYTE  Type;
    WORD  Height;
    WORD  Width;
    WORD  Depth;
    CHAR  FileName[81];
    DWORD Flags;
    BYTE  Filler[32];
} HEADER;

HEADER header;
FILE *fp = fopen("MYFORMAT.FOR", "rb");
if (fp)
    fread(&header, sizeof(HEADER), 1, fp);
```

Here we see a typical bitmap file format header defined as a C language structure. The fields of the header contain information on the size, color, type of image, attribute flags, and name of the image file itself. The fields in the header range from one to 80 bytes in size, and the entire structure is padded out to a total length of 128 bytes.

The first potential gotcha may occur even before you read the file. It lies waiting for you in the *fopen*() function. If you don't indicate that you are opening the graphics file for reading as a binary file (by specifying the "rb" in the second argument of the *fopen*() parameter list), you may find that extra carriage returns and/or linefeeds appear in your data in memory that are not in the graphics file. This is because *fopen*() opens files in text mode by default.

Ex.1007 / Page 140 of 1158
APPLE INC.

In C++, you need to OR the ios::binary value into the mode argument of the fstream or ifstream constructor:

```
fstream *fs = new fstream ("MYFORMAT.FOR", ios::in|ios::binary);
```

After you have opened the graphics file successfully, the next step is to read the header. The code we choose to read the header in this example is the *fread*() function, which is most commonly used for reading chunks of data from a file stream. Using *fread*(), you can read the entire header with a single function call. A good idea, except that in using *fread*() you are likely to encounter problems. You guessed it, the second gotcha!

A common problem you may encounter when reading data into a structure is that of the boundary alignment of elements within the structure. On most machines, it is usually more efficient to align each structure element to begin on a 2-, 4-, 8-, or 16-byte boundary. Because aligning structure elements is the job of the compiler, and not the programmer, the effects of such alignment are not always obvious.

The compiler word-aligns structure elements in the same way. By adding padding, we increased the length of the header so it ends on a 128-byte boundary. Just as we added padding at the end of the header, compilers add invisible padding to structures to do the following:

- Start the structure on a word boundary (an even memory address).
- Align each element on the desired word or doubleword boundary.
- Ensure that the size of the structure is an even number of bytes in size (16-bit machines) or is divisible by four (32-bit machines).

The padding takes the form of invisible elements that are inserted between the visible elements the programmer defines in the structure. Although this invisible padding is not directly accessible, it is as much a part of the structure as any visible element in the structure. For example, the following structure will be five, six, or eight bytes in size if it is compiled using a 1-, 2-, or 4-byte word alignment:

```
typedef struct _test
{
    BYTE  A;    /* One byte */
    DWORD B;    /* Four bytes */
} TEST;
```

With 1-byte alignment, there is no padding, and the structure is five bytes in size, with element B beginning on an odd-byte boundary. With 2-byte alignment, one byte of padding is inserted between elements A and B to allow

Ex.1007 / Page 141 of 1158
APPLE INC.

element B to begin on the next even-byte boundary. With 4-byte alignment, three bytes of padding are inserted between A and B, allowing element B to begin on the next even-word boundary.

Determining the Size of a Structure

At runtime, you can use the *sizeof*() operator to determine the size of a structure:

```
typedef struct _test
{
    BYTE  A;
    DWORD B;
} TEST;
printf("TEST is %u bytes in length\n", sizeof(TEST));
```

Because most ANSI C compilers don't allow the use of *sizeof*() as a preprocessor statement, you can check the length of the structure at compile time by using a slightly more clever piece of code:

```
/*
** Test if the size of TEST is five bytes or not. If not, the array
** SizeTest[] will be declared to have zero elements, and a
** compile-time error will be generated on all ANSI C compilers.
** Note that the use of a typedef causes no memory to be allocated
** if the sizeof() test is true. And please, document all such
** tests in your code so other programmers will know what the heck
** you are attempting to do.
*/
typedef char CHECKSIZEOFTEST[sizeof(TEST) == 5];
```

The gotcha here is that the *fread*() function will write data into the padding when you expected it to be written to an element. If you used *fread*() to read five bytes from a file into our 4-byte-aligned TEST structure, you would find that the first byte ended up correctly in element A, but that bytes 2, 3, and 4 were stored in the padding and not in element B as you had expected. Element B will instead store only byte 5, and the last three bytes of B will contain garbage.

There are several steps involved in solving this problem.

First, attempt to design a structure so each field naturally begins on a 2-byte (for 16-bit machines) or 4-byte (for 32-bit machines) boundary. Now if the compiler's byte-alignment flag is turned on or off, no changes will occur in the structure.

Ex.1007 / Page 142 of 1158
APPLE INC.

When defining elements within a structure, you also want to avoid using the INT data type. An INT is two bytes on some machines and four bytes on others. If you use INTs, you'll find that the size of a structure will change between 16- and 32-bit machines even though the compiler is not performing any word alignment on the structure. Always use SHORT to define a 2-byte integer element and LONG to specify a four-byte integer element, or use WORD and DWORD to specify their unsigned equivalents.

When you read an image file header, you typically don't have the luxury of being the designer of the file's header structure. Your structure must exactly match the format of the header in the graphics file. If the designer of the graphics file format didn't think of aligning the fields of the header, then you're out of luck.

Second, compile the source code module that contains the structure with a flag indicating that structure elements should not be aligned (/Zp1 for Microsoft C++ and -a1 for Borland C++). Optionally, you can put the *#pragma* directive for this compiler option around the structure; the result is that only the structure is affected by the alignment restriction and not the rest of the module.

This, however, is not a terribly good solution. As we have noted, by aligning all structure fields on a 1-byte boundary, the CPU will access the structure data in memory less efficiently. If you are reading and writing the structure only once or twice, as is the case with many file format readers, you may not care how quickly the header data is read.

You must also make sure that whenever anybody compiles your source code, they use the 1-byte structure alignment compiler flag. Depending on which machine is executing your code, failure to use this flag may cause problems reading image files. Naming conventions may also differ for *#pragma* directives between compiler vendors; on some compilers, the byte-alignment *#pragma* directives might not be supported at all.

Finally, we must face the third insidious gotcha—the native byte order of the CPU. If you attempt to *fread*() a graphics file header containing data written in the little-endian byte order on a big-endian machine (or big-endian data in a file on a little-endian machine), you will get nothing but byte-twiddled garbage. The *fread*() function cannot perform the byte-conversion operations necessary to read the data correctly, because it can only read data using the native byte order of the CPU.

Ex.1007 / Page 143 of 1158
APPLE INC.

At this point, if you are thinking, "But, I'm not going to read in each header field separately!" you are in for a rather rude change of your implementation paradigm!

Reading each field of a graphics file header into the elements of a structure, and performing the necessary byte-order conversions, is how it's done. If you are worried about efficiency, just remember that a header is usually read from a file and into memory only once, and you are typically reading less than 512 bytes of data—in fact, typically much less than that. We doubt if the performance meter in your source code profiler will show much of a drop.

So, how do we read in the header fields one element at a time? We could go back to our old friend *fread*():

```
HEADER header;
fread(&header.FileId,   sizeof(header.FileId),   1, fp);
fread(&header.Height,   sizeof(header.Height),   1, fp);
fread(&header.Width,    sizeof(header.Width),    1, fp);
fread(&header.Depth,    sizeof(header.Depth),    1, fp);
fread(&header.Type,     sizeof(header.Type),     1, fp);
fread(&header.FileName, sizeof(header.FileName), 1, fp);
fread(&header.Flags,    sizeof(header.Flags),    1, fp);
fread(&header.Filler,   sizeof(header.Filler),   1, fp);
```

While this code reads in the header data and stores it in the structure correctly (regardless of any alignment padding), *fread*() still reads the data in the native byte order of the machine on which it is executing. This is fine if you are reading big-endian data on a big-endian machine, or little-endian data on a little-endian machine, but not if the byte order of the machine is different from the byte order of the data being read. It seems that what we need is a filter that can convert data to a different byte order.

If you have ever written code that diddled the byte order of data, then you have probably written a set of *SwapBytes* functions to exchange the position of bytes with a word of data. Your functions probably looked something like this:

```
/*
** Swap the bytes within a 16-bit WORD.
*/
WORD SwapTwoBytes(WORD w)
{
    register WORD tmp;
    tmp =  (w & 0x00FF);
    tmp = ((w & 0xFF00) >> 0x08) | (tmp << 0x08);
    return(tmp);
}
/*
** Swap the bytes within a 32-bit DWORD.
*/
```


Ex.1007 / Page 144 of 1158
APPLE INC.

```
DWORD SwapFourBytes(DWORD w)
{
    register DWORD tmp;
    tmp =  (w & 0x000000FF);
    tmp = ((w & 0x0000FF00) >> 0x08) | (tmp << 0x08);
    tmp = ((w & 0x00FF0000) >> 0x10) | (tmp << 0x08);
    tmp = ((w & 0xFF000000) >> 0x18) | (tmp << 0x08);
    return(tmp);
}
```

Because words come in two sizes, you need two functions: *SwapTwoBytes*() and *SwapFourBytes*()—for those of you in the C++ world, you'll just write two overloaded functions, or a function template, called *SwapBytes*(). Of course you can swap signed values just as easily by writing two more functions that substitute the data types SHORT and LONG for WORD and DWORD.

Using our *SwapBytes* functions, we can now read in the header as follows:

```
HEADER header;
fread(&header.FileId,   sizeof(header.FileId),   1, fp);
header.FileId = SwapFourBytes(header.FileId);
fread(&header.Height,   sizeof(header.Height),   1, fp);
header.Height = SwapTwoBytes(header.Height);
fread(&header.Width,    sizeof(header.Width),    1, fp);
header.Width = SwapTwoBytes(header.Width);
fread(&header.Depth,    sizeof(header.Depth),    1, fp);
header.Depth = SwapTwoBytes(header.Depth);
fread(&header.Type,     sizeof(header.Type),     1, fp);
fread(&header.FileName, sizeof(header.FileName), 1, fp);
fread(&header.Flags,    sizeof(header.Flags),    1, fp);
header.Flags = SwapFourBytes(header.Flags);
fread(&header.Filler,   sizeof(header.Filler),   1, fp);
```

We can read in the data using *fread*() and can swap the bytes of the WORD and DWORD-sized fields using our *SwapBytes* functions. This is great if the byte order of the data doesn't match the byte order of the CPU, but what if it does? Do we need two separate header-reading functions, one with the *SwapBytes* functions and one without, to ensure that our code will work on most machines? And, how do we tell at runtime what the byte order of a machine is? Take a look at this example:

```
#define LSB_FIRST       0
#define MSB_FIRST       1
/*
** Check the byte-order of the CPU.
*/
int CheckByteOrder(void)
{
    SHORT  w = 0x0001;
    CHAR  *b = (CHAR *) &w;
    return(b[0] ? LSB_FIRST : MSB_FIRST);
}
```


Ex.1007 / Page 145 of 1158
APPLE INC.

The function *CheckByteOrder*() returns the value LSB_FIRST if the machine is little-endian (the little end comes first) and MSB_FIRST if the machine is big-endian (the big end comes first). This function will work correctly on all big- and little-endian machines. Its return value is undefined for middle-endian machines (like the PDP-11).

Let's assume that the data format of our graphics file is little-endian. We can check the byte order of the machine executing our code and can call the appropriate reader function, as follows:

```
int byteorder = CheckByteOrder();
if (byteorder == LSB_FIRST)
    ReadHeaderAsLittleEndian();
else
    ReadHeaderAsBigEndian();
```

The function *ReadHeaderAsLittleEndian*() would contain only the *fread*() functions, and *ReadHeaderAsBigEndian*() would contain the *fread*() and *SwapBytes*() functions.

But this is not very elegant. What we really need is a replacement for both the *fread*() and *SwapBytes* functions that can read WORDs and DWORDs from a data file, making sure that the returned data is in the byte order we specify. Consider the following functions:

```
/*
** Get a 16-bit word in either big- or little-endian byte order.
*/
WORD GetWord(char byteorder, FILE *fp)
{
    register WORD w;
    if (byteorder == MSB_FIRST)
    {
        w =  (WORD) (fgetc(fp) & 0xFF);
        w = ((WORD) (fgetc(fp) & 0xFF)) | (w << 0x08);
    }
    else          /* LSB_FIRST */
    {
        w  =  (WORD) (fgetc(fp) & 0xFF);
        w |= ((WORD) (fgetc(fp) & 0xFF) << 0x08);
    }
     return(w);
}

/*
** Get a 32-bit word in either big- or little-endian byte order.
*/
```


Ex.1007 / Page 146 of 1158
APPLE INC.

```
DWORD GetDword(char byteorder, FILE *fp)
{
    register DWORD w;

    if (byteorder == MSB_FIRST)
    {
        w =  (DWORD) (fgetc(fp) & 0xFF);
        w = ((DWORD) (fgetc(fp) & 0xFF)) | (w << 0x08);
        w = ((DWORD) (fgetc(fp) & 0xFF)) | (w << 0x08);
        w = ((DWORD) (fgetc(fp) & 0xFF)) | (w << 0x08);
    }
    else          /* LSB_FIRST */
    {
        w |=   (DWORD) (fgetc(fp) & 0xFF);
        w |= (((DWORD) (fgetc(fp) & 0xFF)) << 0x08);
        w |= (((DWORD) (fgetc(fp) & 0xFF)) << 0x10);
        w |= (((DWORD) (fgetc(fp) & 0xFF)) << 0x18);
    }
    return(w);
}
```

The *GetWord*() and *GetDword*() functions will read a word of data from a file stream in either byte order (specified in their first argument). Valid values are LSB_FIRST and MSB_FIRST.

Now, let's look at what reading a header is like using the *GetWord*() and *GetDword*() functions. Notice that we now read in the single-byte field Type using *fgetc*() and that *fread*() is still the best way to read in blocks of byte-aligned data:

```
HEADER header;

int byteorder = CheckByteOrder();

header.FileId   = GetDword(byteorder, fp);
header.Height   = GetWord(byteorder, fp);
header.Width    = GetWord(byteorder, fp);
header.Depth    = GetWord(byteorder, fp);
header.Type     = fgetc(fp);
fread(&header.FileName, sizeof(header.FileName), 1, fp);
header.Flags    = GetDword(byteorder, fp);
fread(&header.Filler, sizeof(header.Filler), 1, fp);
```

All we need to do now is to pass the byte order of the data being read to the *GetWord*() and *GetDword*() functions. The data is then read correctly from the file stream regardless of the native byte order of the machine on which the functions are executing.

The techniques we've explored for reading a graphics file header can also be used for reading other data structures in a graphics file, such as color maps, page tables, scan-line tables, tags, footers, and even pixel values themselves.

Ex.1007 / Page 147 of 1158
APPLE INC.

Reading Image Data

In most cases, you will not find any surprises when you read image data from a graphics file. Compressed image data is normally byte-aligned and is simply read one byte at a time from the file and into memory before it is decompressed. Uncompressed image data is often stored only as bytes, even when the pixels are two, three, or four bytes in size.

You will also usually use *fread*(), to read a block of compressed data into a memory buffer that is typically 8K to 32K in size. The compressed data is read from memory a single byte at a time, is decompressed, and the raw data is written either to video memory for display, or to a bitmap array for processing and analysis.

Many bitmap file formats specify that scan lines (or tiles) of 1-bit image data should be padded out to the next byte boundary. This means that if the width of an image is not a multiple of eight, then you probably have a few extra zeroed bits tacked onto the end of each scan line (or the end and/or bottom of each tile). For example, a 1-bit image with a width of 28 pixels will contain 28 bits of scan-line data followed by four bits of padding, creating a scan line 32 bits in length. The padding allows the next scan line to begin on a byte boundary, rather than in the middle of a byte.

You must determine whether the uncompressed image data contains scan-line padding. The file format specification will tell you if padding exists. Usually, the padding is loaded into display memory with the image data, but the size of the display window (the part of display memory actually visible on the screen) must be adjusted so that the padding data is not displayed.

For Further Information About Reading Graphics Data

An excellent article concerning the solution to byte-order conversion problems appeared in the now-extinct magazine, *C Gazette.* It was written by a fellow named James D. Murray!

Murray, James D., "Which Endian is Up?" *C Gazette,* Summer, 1990.

Writing Graphics Data

As you might guess, writing a graphics file is basically the inverse of reading it. Writers may send data directly to an output device, such as a printer, or they may create image files and store data in them.

Ex.1007 / Page 148 of 1158
APPLE INC.

Writing a Graphics File Header

When you write a graphics file header, you must be careful to initialize all of the fields in the header with the correct data. Initialize reserved fields used for fill and padding with the value 00h, unless the file format specification states otherwise. You must write the header data to the graphics files using the correct byte order for the file format as well.

Because the *GetWord*() and *GetDword*() functions were so handy for correctly reading a header, their siblings *PutWord*() and *PutDword*() must be just as handy for writing one:

```
/*
** Put a 16-bit word in either big- or little-endian byte order.
*/
void PutWord(char byteorder, FILE *fp, WORD w)
{
    if (byteorder == MSB_FIRST)
    {
        fputc((w >> 0x08) & 0xFF, fp);
        fputc( w          & 0xFF, fp);
    }
    else          /* LSB_FIRST */
    {
        fputc( w          & 0xFF, fp);
        fputc((w >> 0x08) & 0xFF, fp);
    }
}

/*
** Put a 32-bit word in either big- or little-endian byte order.
*/
void PutDword(char byteorder, FILE *fp, DWORD w)
{
    if (byteorder == MSB_FIRST)
    {
        fputc((w >> 0x18) & 0xFF, fp);
        fputc((w >> 0x10) & 0xFF, fp);
        fputc((w >> 0x08) & 0xFF, fp);
        fputc( w          & 0xFF, fp);
    }
    else          /* LSB_FIRST */
    {
        fputc( w          & 0xFF, fp);
        fputc((w >> 0x08) & 0xFF, fp);
        fputc((w >> 0x10) & 0xFF, fp);
        fputc((w >> 0x18) & 0xFF, fp);
    }
}
```

In the following example, we use *fwrite*(), *PutWord*(), and *PutDword*() to write out our header structure. Note that the byteorder argument in *PutWord*() and

Ex.1007 / Page 149 of 1158
APPLE INC.

PutDword() indicates the byte order of the file we are writing (in this case, little-endian), and not the byte order of the machine on which the functions are being executed. Also, we indicate in *fopen*() that the output file is being opened for writing in binary mode (wb):

```
typedef struct _Header
{
    DWORD FileId;
    BYTE  Type;
    WORD  Height;
    WORD  Width;
    WORD  Depth;
    CHAR  FileName[81];
    DWORD Flags;
    BYTE  Filler[32];
} HEADER;

int WriteHeader()
{
    HEADER header;
    FILE *fp = fopen("MYFORMAT.FOR", "wb");

    if (fp)
    {
        header.FileId   = 0x91827364;
        header.Type     = 3;
        header.Depth    = 8;
        header.Height   = 512;
        header.Width    = 512;
        strncpy((char *)header.FileName, "MYFORMAT.FOR",
       sizeof(header.FileName));
        header.Flags    = 0x04008001;
        memset(&header.Filler, 0, sizeof(header.Filler));

        PutDword(MSB_FIRST, fp, header.FileId);
        fputc(header.Type, fp);
        PutWord(MSB_FIRST, fp, header.Height);
        PutWord(MSB_FIRST, fp, header.Width);
        PutWord(MSB_FIRST, fp, header.Depth);
        fwrite(&header.FileName, sizeof(header.FileName), 1, fp);
        PutDword(MSB_FIRST, fp, header.Flags);
        fwrite(&header.Filler, sizeof(header.Filler), 1, fp);

        fclose(fp);
        return(0);
    }
    return(1);
}
```


Ex.1007 / Page 150 of 1158
APPLE INC.

Writing Image Data

Writing binary data can be a little more complex than just making sure you are writing data in the correct byte order. Many formats specify that each scan line is to be padded out to end on a byte or word boundary if it does not naturally do so. When the scan-line data is read from the file, this padding (usually a series of zero bit values) is thrown away, but it must be added again later if the data is written to a file.

Image data that is written uncompressed to a file may require a conversion before it is written. If the data is being written directly from video memory, it may be necessary to convert the orientation of the data from pixels to planes, or from planes to pixels, before writing the data.

If the data is to be stored in a compressed format, the quickest approach is to compress the image data in memory and use *fwrite*() to write the image data to the graphics file. If you don't have a lot of memory to play with, then write out the compressed image data as it is encoded, usually one scan line at a time.

Test Files

How can you be sure that your application supports a particular file format? Test, test, test...

Files adhering to the written format specification, and software applications that work with them, are called *fully conforming*, or just *conforming*. Fully conforming software should always make the conservative choice if there is any ambiguity in the format specification. Of course it's not always clear what conservative means in any specified context, but the point is to not extend the format if given the chance, no matter how tempting the opportunity. In any case, if you write software to manipulate graphics files, you'll need some fully conforming files to test the code.

If you happen to be working with TIFF, GIF, or PCX, files in these formats are often just a phone call away, but beware. Files are not necessarily fully conforming, and there is no central registry of conforming files. In some cases, effort has gone into making sets of canonical test files; for example, some are included in the TIFF distribution.

The trick is to get a wide spectrum of files from a number of sources and to do extensive testing on everything you find—every size, resolution, and variant that can be found. Another strategy is to acquire files created by major applications and test for compatibility. For example, Aldus PageMaker files are often used to test TIFF manipulation software. Unfortunately, PageMaker has

Ex.1007 / Page 151 of 1158
APPLE INC.

historically been the source of undocumented de facto TIFF revisions, and testing compatibility with PageMaker-produced TIFF files does not produce wholly accurate results.

Any number of graphics file conversion programs exist that can do screen captures, convert files, and display graphics and images in many different formats. Once again, you are at the mercy of the diligence of the application author.

Remember, too, that it's your responsibility to locate the latest format revisions. We still see new readers, for instance, on all platforms, which fail to support the latest specification. If you simply cannot locate an example of the format you need, your last resort will always be the format creator or caretaker. While this approach is to be used with discretion, and is by no means guaranteed, it sometimes is your only recourse, particularly if the format is very new.

In any case, don't stop coding and testing simply because one variation of an image file format can be read and converted successfully by your application. Test your application using as many variations of the format, and from as many different sources as possible. Have your application do its best to support the image format fully.

Corruption of Graphics Files

What is a corrupt graphics file?

Imagine every munged image you've ever seen displayed by a graphical display program—shifted or broken pictures, large areas of snowy or mosaic patterns, colors that only seriously appear in Andy Warhol paintings, or simply no image displayed at all. Why does this happen?

Causes of Corruption

When an image fails to display properly, the cause(s) might be any or all of the following:

- The display environment (drivers, video memory, display resolution) is not properly configured or is inadequate.
- The file format is not supported by the display program.
- This variation of the file format is not supported by the display program.
- The display program is incorrectly interpreting the file.
- The file data is bad or corrupt

Ex.1007 / Page 152 of 1158
APPLE INC.

Problems with the display environment

Most graphical display problems can be corrected by adjusting one or more aspects of the display environment. For example, if you attempt to display a truecolor image using a video graphics card or software driver that does not support the full bitdepth or resolution of the image, the display program will either reduce the number of colors in the displayed image, or simply refuse to display the image at all. In either case, the results will probably not look as you expected.

Installing the proper software driver for the graphics card, resolution, and number of colors desired may fix this display problem. Upgrading your display program to a newer version or using a different program is also an option. And, of course, who couldn't use a faster graphics card with more memory, and a larger, higher-resolution display monitor as well?

Sometimes the fault lies with the file reader. A display program should make every attempt to verify that it understands the format of the data it is reading. For example, reading a JPEG file as a GIF file may cause a display program to produce unexpected results.

Many file formats have different internal variations depending upon the revision level of the format and the type of data that the files store. Most formats (e.g., TIFF and TGA) make it easy to determine the type of data stored in the file by looking at header information, and others (e.g., PCX) make it somewhat more difficult. Some formats (e.g., Amiga IFF) use a different file extension for every type of data that they store, and others (e.g., XBM) store only one type of data.

You're asking for trouble if you assume from a file's extension that the file has a specific format. More than one file has been given an improper file extension. And if the display program is accepting input directly from a data stream, there won't be a file extension to read in any case.

With most formats, the file extension doesn't change to reflect the revision level of the file format, or the type of data stored in the file. This lack of human-readable recognition has probably hurt the TIFF file format the most. A TIFF file can store any type of image data ranging from monochrome to truecolor, and can compress it using any one of a half-dozen or more methods of encoding.

Many TIFF viewers support RLE-compressed or uncompressed monochrome images, but not images compressed using CCITT G3 and G4 encoding. Other TIFF viewers support gray-scale and palette color images but not truecolor images. Some viewers support the display of images compressed using the JPEG or TIFF-LZW algorithms, and (most) others do not.

Ex.1007 / Page 153 of 1158
APPLE INC.

As you can see, many different types of images can be stored in a TIFF file—and all of them will have the ".TIF" extension. It's no wonder that users become frustrated when some TIFF files display and others don't; after all, in a directory listing all the files "look" the same. Make sure that not all files look the same to your file reader.

Problems with the program code

We've discussed the fact that an image might fail to display properly because the program reading it lacks the features to display the image data. It is also quite possible that an image may fail to display because of bugs in the display program's code. Such bugs usually result from the fact that the programmer misinterprets the file format specification, or that he doesn't fully understand the programming language that he's using, or that he includes some other programmer's buggy code in his own code. Testing the display program on a wide variety of image files will reveal such problems.

Problems with the data

What about the graphics file? Can the file itself be the source of the problem?

A graphics file is no different from any other type of data file on your system. Like other files, the data a graphics file contains may be incorrectly constructed (bad data) or damaged (corrupt data). Bad or corrupt files occur as a consequence of one or more of the following problems:

- Buggy file writer
- Uncorrected transmission error
- Bad write to disk
- Faulty processing

Bad data may result from a poorly designed file format writer. Improperly calculating header values (e.g., number of colors, resolution, file size, etc.) or writing the data in an incorrect byte order will mislead a format reader into incorrectly interpreting the image file data. Buggy codecs (encoder/decoders) may produce badly encoded data that may appear to be doing its job but actually violates the compression algorithm. (For example, the TIFF specification implements the LZW algorithm in a faulty way.)

Files may become corrupted in a variety of ways. For example:

- Transferring files between computer systems using a serial or network transmission protocol that doesn't do error detection and correction invites the possibility of errors due to line noise or lost data.

Ex.1007 / Page 154 of 1158
APPLE INC.

- Copying a file to a damaged filesystem or disk may result in write errors that destroy part of the file.
- An improperly handled software exception may write data to random parts of the filesystem, thus corrupting any files touched by the write.
- Sending a binary file through a 7-bit data channel, or a text filter that converts carriage returns to linefeeds, guarantees permanent damage to the file's data.

Detecting File Corruption

File format readers must be able to quickly detect that a file's data is incorrect or unexpected, or that it in some way violates the specification of the file format or data compression algorithm. Quick detection allows the reader to respond with an error message to the user, and prevents the untimely crash of the program by the reading of bad data. An accurate analysis of the problem by the file reader and a verbose error message displayed to the program user are also required.

How can you tell whether a file is corrupt? We describe several indicators below.

EOF marker

The end of file (EOF) file stream marker is a good indicator. Often, graphics files are truncated through errors in transmission or by a failed write operation to a disk. In such cases, when the file is read, the EOF will occur much sooner than a file format reader would have expected, and corruption of the file may be assumed. Read operations will also fail if there is an actual error in the filesystem or disk. Always check the return value of your read operations. An unexpected EOF, or any file stream error, is a sure sign that something is wrong.

Unexpected characters

Missing or excessive data may cause an improper alignment of the internal structures of a file format. Data structures in memory often contain invisible 2- or 4-byte boundary padding between structure elements that may unintentionally be written to a file. Data written to a file opened in text mode, rather than in binary mode, may contain embedded carriage return and/or linefeed characters and may therefore create bad data.

Ex.1007 / Page 155 of 1158
APPLE INC.

Magic value errors
Stream-oriented formats divide stored data into individual sections called *segments* (blocks, chunks, etc.), each of which begins with a specific identification or "magic" value followed by the length of the data in the segment. If a format reader reads in an entire segment and discovers that the next data in the file is not the expected magic value of the following segment (or the end of data stream marker), then the reader assumes that the data is bad or corrupt.

Out-of-range offset values
File-oriented formats typically use fixed-size data structures and absolute offset values to locate data. An offset value that points outside the file space is a sure indication that the offset value is wrong, or the file has been truncated.

Hints for Designing File Readers and Writers

What should a file format reader do in the case of missing or excessive data? It depends on the file format and the data itself. If the bad information is trivial (e.g., a text comment or a thumbnail image), the reader may choose to ignore the bad data and continue reading the file. If the information is critical (e.g., the header), the reader should simply give up.

Regardless of the action it takes, a file reader should display a warning, error, or diagnostic message to indicate that something unexpected has occurred. Messages such as "Unknown file format", "Unknown compression type", "Unsupported resolution", or "Corrupt data" will at least give the user a clue as to what is wrong.

Here are some tips for designing a error-detecting file format reader:

- Always check input operations for file stream read errors and end of file indications.
- Check header field data against an expected range of value. Don't use data that seems unreasonable.
- Don't trust the file extension to identify the file format. Identify the file format by the data expected in the header.
- Always calculate and compare checksum and CRC values included in any image files used to verify the integrity of the data.
- Use a binary editor to damage a number of graphics files and see how your reader handles them. Uuencoding a graphics file, adding, deleting, or changing a few lines, and then uudecoding the file is also an easy way to corrupt a graphics file. A reader should never crash when reading a bad file.

Ex.1007 / Page 156 of 1158
APPLE INC.

Here are some tips for designing a file format writer:

- Always check output operations for file stream write errors.
- Store file and data structure sizes in the file header, if supported.
- Use any built-in methods of error detection, such as a CRC value, that the format may support.
- Use other display programs to check your generated files.

Most file formats do not have built-in mechanisms for error detection. Instead, we rely on the file reader to recognize bad or corrupt data, based on information stored in the file, and to react accordingly. Some formats store the size of the graphics data, or even the length of the entire file, in their headers. Other formats contain fixed-sized data structures that change only between revisions of the format. These features are not specifically designed to detect or correct file or data errors, but they can be used in that way and are better than nothing.

At least one format, PNG, does include an active error-checking mechanism. PNG is a data stream format that comprises a small signature of byte values followed by three or more chunks of data. Each of these chunks stores a 4-byte CRC-32 value calculated from the data in the chunk. A PNG reader can calculate the CRC of the data and then compare this value to the one stored in the chunk. If the values do not match, the reader can assume that the data in that chunk is corrupt.

The PNG signature is also unique in that it contains several characters used to detect whether the file was improperly processed by a 7-bit data channel or a text processing filter. (See the PNG article in Part Two of this book for more information.)

For formats that don't provide any real error checking, you might consider storing files using a file archiving program that offers error checking as a feature. Many archivers, such as *pkzip* and *zoo*, perform a CRC calculation on each file that they store. When the file is removed from the archive, the value is recalculated and compared to the stored value. If they match, then you know that the file has not been corrupted. Archiving your graphics files is especially recommended if you are sending them over a data communications network, such as the Internet.

Another type of external error-detecting mechanism is a digital signature. This is a method of detecting whether changes have occurred within a block of information. We discuss digital signatures in the context of graphics file encryption in the section that follows.

Ex.1007 / Page 157 of 1158
APPLE INC.

Encryption of Graphics Files

Cryptography is the technology of keeping information secret. In this context, we define secret as "being protected from unauthorized access and attack." Although you may not think of your graphics files or their contents as ever being under attack, you may want to keep the information contained in these files from being copied or viewed by unauthorized people or computers. If copies of the files are freely available, the only way to keep the files secret is to encrypt them.

Cryptography may seem to be a black art requiring extremely complex mathematics and access to supercomputers. This may be the case for professional cryptanalysts (codebreakers). But for ordinary people who need to protect data, cryptography can be a strong, often simple to use, and sometimes freely available tool.

This section doesn't try to explain cryptography, nor the details of particular cryptosystems. (Refer to the books in the "For Further Information" section for basic information in this area.) Instead, we will look at why you, as an author, archiver, transporter, or user of graphics files, may need to encrypt graphics files.

First, let's look at the general problem of protecting graphics files.

Protecting Graphics Files

Many businesses and organizations, such as those associated with medical and document imaging, worry about users (most often programmers) finding ways to alter the contents of their graphics and image files. Such alterations might be used to change an X-ray photograph, discover a debit card number, or forge a handwritten signature on the bitmap of a check. In addition to the human consequences, such actions may result in lawsuits against the company whose process or equipment originally created the graphics file—not a pleasant prospect for any corporation.

How can you protect against alterations of this kind?

Physical protection

The initial defense against unauthorized alterations is to deny physical access to your graphics files. How would you do this? When a new image is saved, store it in a file that is physically and directly inaccessible to the user. When an image is needed, decrypt (and possibly decompress) it, then display or print it, but never make the file itself directly accessible.

Ex.1007 / Page 158 of 1158
APPLE INC.

Unfortunately, this approach is not feasible for operations requiring that the image data be transmitted along unsecured channels, or possibly stored on a unsecured system. In such cases, we must instead find a way to make the contents of our graphics files secure while allowing the files to be stored on any computer system, even one that is not secure.

Proprietary file formats

Files can be difficult, if not impossible, to read if you do not know the format in which the file's data is stored. If we use a well-published graphics file format to store our data, a programmer could trivially write a program that makes unauthorized use of our data. So, it would seem that creating a proprietary graphics file format, one whose internals are kept secret, would go a long way toward protecting our data. The problem with this method is that the skill and determination of the people who want to view and alter your graphics files may be much greater than your own.

Even general information can be enough on which to base an attack. For example, somebody who knows that your file is a bitmap will also know that the vast majority of bitmap files contain a fixed-size header followed by the image data. All bitmap headers contain typical information, such as the size of the bitmap and the number of bits per pixel. Inventing a header from scratch (or worse, basing your new format on an already well-known format) and following it with data encoded using a conventional data compression scheme (or worse, no compression at all) will not slow down a determined file format cracker for very long. The cracking of Kodak's Photo CD format, for example, came about through the exploitation of a very few bits of general information.

We can assume that the most useful contents of even an unknown graphics file header can be deduced. What about the image data itself? Bitmap data is usually stored as pixels that are 1, 8, or 24 bits in size. The pixel values are typically indexed using a color table, or stored directly using a three-color model, such as RGB or CMY. Only a few formats store bitmap data otherwise, so discovering the format of the bitmap may be just as easy as deducing the contents of the header.

Proprietary compression methods

What about using compression as a means of obscuring data? Certainly a bitmap can be compressed, and compression does obscure the apparent format of the bitmap. But once the starting byte of the bitmap is identified, it is a simple matter to try any of a dozen or so common algorithms to decompress the data. It may even be possible to identify the type of compression method

Ex.1007 / Page 159 of 1158
APPLE INC.

used by looking at a hex dump of a section of the bitmap that is a single color. And, although the scan lines of the bitmap may not be stored in sequential order (such as in interlaced GIF files), that will not prevent the method of compression from being discovered.

One way to get around this is to use a compression method that a file format cracker won't have in her toolbox. Unfortunately, data compression algorithms are designed only to make your data physically smaller, and not to secure your data from prying eyes (or in-circuit emulators). Of course, a few unpublished, proprietary methods of compression, such as fractal compression, are also considered forms of data encryption because the details are kept secret. These methods derive their security from their complexity, however, not from the use of robust algorithms. It's only a matter of time before they are cracked. Also, note that these are extremely complex encoding algorithms; if you don't know how to decompress the data, you certainly can't use it in a practical way.

It is doubtful that the average business—or even the above-average programmer—has the resources to invent a method of compression that is radically different from, yet just as good as, JPEG, LZW, or CCITT Group 4. You can alter an existing, published compression algorithm to "break" it in an undocumented way, thereby rendering it undecodable by conventional means. However, doing so risks decreasing the efficiency of the compression method and leaving your files larger in size, and possibly slower to decompress, than they otherwise would be. You also risk giving yourself a false sense of security.

In summary, proprietary file formats and bizarre compression methods are not secure methods of protecting your data. We recommend that you consider encryption instead.

Why Encrypt Graphics Files?

There are a number of possible reasons why you might want to encrypt your graphics files:

- **To hide the contents of the file from unauthorized users.** Most people who are interested in file encryption will answer this way. Suppose that you have some drawings or images that you want to keep most people from seeing. Simply hiding the files does not provide enough security for your purposes. Or, you may want a second line of defense in case somebody does discover your hidden files. Often, keeping your graphics files inaccessible by locking them up isn't an option anyway, because the files must be distributed.

Ex.1007 / Page 160 of 1158
APPLE INC.

- **To prevent the file from being illegally copied.** Encryption schemes can't actually prevent files from being copied, but they can help ensure that unauthorized copies of files are useless to the person who copies them. Files distributed on a CD-ROM are often encrypted; they are useless until a key is used to decrypt them. The key is usually given to the user when he registers the product. Encrypted files transmitted via modem are still vulnerable to wiretapping, but without the key they are useless to the wiretapper. (Of course, remember that if the key is discovered as well, the files may be decrypted.)
- **To disguise the type of file.** This case is a bit esoteric, but it's possible that you may not want others to know what kind of data the file contains. Obscuring not only the file's data, but also the type of data the file contains, adds a small bit of additional difficulty to cracking the file via certain cryptanalytic attacks.
- **To detect whether a file is corrupt.** Corruption of encrypted files is easily detected when the files are decrypted. Any alteration in the encrypted data, even to the extent of changing only a single byte, causes the decryption process to fail. However, this method can only indicate that the file is corrupt, but not specifically where in the file the corruption occurred. For this purpose, digital signatures or simple checksums work better.

  Corruption within a file can also be detected when a file is decompressed. Most graphics files, however, are written in a format that supports only the compression of bitmap data. Corrupt data in uncompressed parts of the file, such as the color tables and the header, would go undetected. Many formats (e.g., BMP) routinely store bitmap data uncompressed; others (e.g., most vector formats) do not support compression at all.

  While you may opt for a simpler method of data error detection in your graphics files (such as storing your files using an archiving utility that supports error detection, as we discuss earlier in this chapter in the section called "Corruption of Graphics Files"), you might also find encryption helpful for this purpose, especially if you are not at liberty to alter the file's contents.
- **To prevent the contents of the file from being altered.** Encryption cannot actually prevent a file from being modified; only the security of a file storage system can do that. But, as we've discussed, encrypted files that have been changed are easily detected. Encrypting your files and publicizing this fact may also provide a deterrent that keeps unauthorized people from attempting to modify your graphics files, or even attempting to locate your

Ex.1007 / Page 161 of 1158
APPLE INC.

hidden files in the first place. Of course, advertising such a fact may present a clear challenge to certain people. (After all, if your program can decode your data, then such people may have a shot at cracking your encryption by examining the internals of your software.)

- **To identify the person or program that created the file.** Encryption can also be used to create a digital signature or "fingerprint" associated with a particular file. A *digital signature* not only verifies the person or program that created the file, but can also include a time stamp of when the signature was created. Most encryption systems allow you to generate a digital signature without actually encrypting the data.

Pros and Cons of Cryptography

Before we look in any more detail at what cryptography is, let's discuss what it is not.

Misconceptions about cryptography

Cryptography is not a form of data compression. Many cryptographic systems do use data compression algorithms to compress your data before they encrypt it. However, this step is often performed not only to reduce the physical size of the data, but also to remove redundancies in the data that might make the file easier to crack. This is the reason why many encrypted files are physically smaller than the original unencrypted files.

Cryptography is not copy protection. Encrypted files may be copied from CD-ROMs, floppies, and hard disks just as easily as any other files can be. Copy protection schemes usually physically alter the media the files are stored on, or format the media in some non-standard way (such as Microsoft's DMF disk format). And while some copy protection schemes may make use of file encryption, it is not the encryption itself that prevents the files from being copied. In this case, encryption can only ensure that if a file is copied, that file will be unusable.

Benefits of cryptography

What are the benefits of using cryptography?

- Cryptography is specifically designed to allow you to get at your data while preventing unauthorized people from doing so as well.
- Encryption allows you to work with standard graphics files as your unencrypted data. This eases the burden on the authorized users of the file. It also releases you from the foolish and quixotic task of inventing yet another proprietary file format. Inventing a proprietary format will add nothing to the security of your encrypted file anyway.

Ex.1007 / Page 162 of 1158
APPLE INC.

- The encrypted files need not be hidden or copy-protected to be secure. If you choose, the file may be freely available for copying and access.
- Encryption algorithms are much harder to crack than either proprietary file formats or data compression algorithms. It is also less likely that unauthorized people will even attempt to obtain your files if they know that they are encrypted.
- Your files are secure for as long as the encryption algorithm itself remains unbroken and your decryption keys remain undiscovered.

These are all very good reasons to use cryptographic technology to secure your data. Are there any reasons not to use encryption? What problems could there be with encrypting your graphics files? To answer this we have to briefly look at two basic systems of cryptography called *private key cryptography* and *public key cryptography*.

Private and public key cryptography

If you have ever used a secret password to log into a computer system or network, withdraw money from an automatic teller machine, or gain entrance to a secret meeting, you have probably used a private key cryptographic system.

Private key systems use a single password or pass phrase to both encrypt and decrypt a piece of information. Both the person encrypting the information and all of the people authorized to decrypt the information must have the key. If an unauthorized person acquires both the encrypted information and the key, and if they know the encryption algorithm that is being used, they can easily access the your information.

Public key systems use an algorithm to generate two mathematically related keys. Data encrypted with one key (the public key) can only be decrypted with the other key (the secret key) and the secret key's pass phrase. The data is secure if no one else has a copy of your secret key and knows your key's pass phrase (and if you don't leave decrypted copies of your files around).

Public key systems have some tremendous advantages over private key systems. Suppose you transmit some private-key-encrypted files to a friend across the country. Your friend can't decrypt these files until you also give her the same password you used to encrypt them. But, how do you securely tell her what the password is? If the password is intercepted along with your files, your data will be in unauthorized hands.

If, on the other hand, you use a public key system, you only need your friend's freely available public key to encrypt the files. Once your friend receives the

Ex.1007 / Page 163 of 1158
APPLE INC.

files, she uses her own secret key and her secret key's pass phrase to decrypt them. No secret pass phrase need be sent with the files. Moreover, your friend's public key (and indeed your own) need not be hidden. In fact, you want as many people to have your public key as possible so they will be able to send encrypted files to you!

Risks of cryptography

Cryptography is not without its risks and problems:

- The more people who know your password, or pass phrase, and have access to your secret keys, the less secure your data is. This is even more true if your encrypted data is easily available and the method of encryption is widely known.
- Even if your keys and pass phrases are secure, your chosen method of encryption may not be. In theory, for every data encryption algorithm invented, at least one method of getting around it exists, although it may take hundreds of computers and many years to fully exploit that method.
- So too, any given implementation of a cryptographic system may not be ideally secure. Even the most theoretically robust of encryption methods, if implemented using bad design and with buggy code, can be insecure.
- Encryption always imposes a performance penalty. Public key systems are especially slow to encrypt and decrypt data. You may decide that your application can't afford the extra overhead to use a particular method of encryption.
- Almost all forms of encryption systems are patented. Even those that are freely available still require a license for commercial use.
- Many import and export restrictions also exist that could make it more difficult to distribute and sell your software.*

Assuming that you can live with these shortcomings of encryption systems (or perhaps you simply want to play around with some of the technology), how can you actually encrypt your files? One tool that's easy to get and use is PGP (Pretty Good Privacy).

* (Simson Garfinkel's *PGP: Pretty Good Privacy*, referenced in "For Further Information" below, contains a complete discussion of patent and export issues.)

Ex.1007 / Page 164 of 1158
APPLE INC.

Using PGP to Encrypt Graphics Files

PGP is a robust public key system, invented by Phil Zimmermann, which is used to securely encrypt and decrypt files. PGP is also a customizable software tool that is capable of creating and managing public and private keys, creating digital signatures, and being integrated into software programs, such as text and graphics file editors and email applications. PGP is also freely available for non-commercial use.

How would you use PGP to encrypt a graphics file? This depends upon whether you wish the resulting encrypted data to be stored as binary or ASCII data. Using the following command, PGP will store encrypted data to a file as binary data:

```
unix% pgp -c private.gif
```

This command causes PGP to read the file *private.gif* and to create a new file called *private.pgp*. The contents of *private.pgp* are a spew of binary data that is an encrypted representation of the data stored in *private.gif*. If you look at *private.pgp* in a text editor, you will see unintelligible binary data.

If you prefer that the encrypted data be stored in ASCII character format, PGP can produce the encrypted file using ASCII character data:

```
unix% pgp -ca private.gif
```

PGP now creates a file called *private.asc*. This file contains an encrypted version of *private.gif* using only ASCII characters. If you look at *private.asc* via a text editor, you will see something that looks like this:

```
-----BEGIN PGP MESSAGE-----
Version: 2.6

pgAAAm5mv5vjsqw+3E+nZvfweBVh8+h4xueb3LnyKgyDjReGzxdvfZcyPQQCXrb9
yegMQoPL9yprzA0cKq0REhlWsE1tq02bFZXDTNjEfDZ5X/ucTutFS4M6fQJFCMWA
R3RC2HwdW8KUZ8/qRK6sf1hisx/HBOuddcjiI+5gj7GFitoXYIu5f/i9wfQlXiqy
Z9NLaPmcROjS9p695s1nkVVA1Z97Dy8f/gswsmg82cQ2RJ51KzHJaINWMOvZWHkz
epnn
=jE5E
-----END PGP MESSAGE-----
```

Instead of binary gibberish, you will see your encrypted file encoded as ASCII data using the radix-64 binary-to-ASCII encoding algorithm. Using this algorithm, an encrypted file stored in ASCII format is one-third larger than the same file encrypted and stored using the binary format. This is because radix-64 encodes every three binary characters as four ASCII characters. This result is well-known to anyone who has used the uuencode program native to UNIX.

Ex.1007 / Page 165 of 1158
APPLE INC.

What does this mean to you? You can store your graphics files as binary or ASCII data. The contents of the files will be hidden. In fact, anyone looking at the encrypted data won't be able to tell what kind of files they are. If the files are altered or corrupted in any way, they will not properly decode. And without the matching secret key and pass phrase, the file's contents cannot be successfully decrypted.

This way, you can keep people from discovering a file's contents, detect whether an encrypted file has been changed, and even hide the type of data a file contains. But how do you use PGP to verify who created the file? PGP does this by using a digital signature.

A digital signature is a numerical value created using an algorithm known as a *message digest function.* This function reads your file data as input and generates a value that is unique to the data in the file. Changing so much as one bit in the file will cause the digital signature to be different.

Digital signatures are typically used to authenticate the sender of an email message. A unique digital signature is created from the file data, encrypted with the private key of the sender, and appended to the message. The message receiver then obtains the public key of the sender, decrypts the digital signature, and calculates the digital signature of the message. If the calculated digital signature matches the signature included in the message, the receiver may safely assume that the message was sent by the owner of the public key.

You can take any graphics file and sign it using PGP. The signature may be used to verify the creator (human or computer) of the file, and the file's contents need not be encrypted. Here's how you would do it using PGP:

```
unix% pgp -s private.gif
```

This command signs the file *private.gif* using your secret key and pass phrase. The encrypted signature is appended to the data and stored in the file *private.pgp.* The data of the file is not encrypted. If you want the file's data to be encrypted, use the following command instead:

```
unix% pgp -se private.gif
```

This command encrypts the file data, signs the file contents, and also places the data in the file *private.pgp.* The signature is a binary value by default. If you want an ASCII signature, specify:

```
unix% pgp -sea private.gif
```


Ex.1007 / Page 166 of 1158
APPLE INC.

This command creates a file called private.asc, which contains the encrypted data and an appended ASCII signature. If the input file contains text rather than binary data, you add yet another flag:

```
unix% pgp -seat private.dxf
```

NOTE

PGP is a self-contained software program that has been ported to many different computers and operating systems. PGP is typically used from the command line. A software application using PGP version 2.*x* could also use PGP as a separate executable program. PGP version 3.0 promises an API library of tools that may be directly linked into software applications.

See the PGP reference in the section below for complete information about this program and all its options.

For Further Information About Encryption

The Internet abounds with information about data encryption. A Web search on the keywords "encryption" and "cryptography" will turn up hundreds of hits.

Two major sources of information and references are the USENET newsgroups *alt.security.pgp* and *sci.crypt* and their associated FAQs.

In addition, see the following references for information about encryption:

Garfinkel, Simson, *PGP: Pretty Good Privacy*, O'Reilly & Associates, Sebastopol, CA, 1995.

Schneier, Bruce, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, New York, NY, 1994.

Schneier, Bruce, *Practical Cryptography*, John Wiley & Sons, New York, NY, 1994.

Schneier, Bruce, "Untangling Public Key Cryptography," Dr. Dobb's Journal, May 1992, pp. 16–28.

Schneier, Bruce, "The IDEA Encryption Algorithm," Dr. Dobb's Journal, December 1993, pp. 50–56.

Stevens, A., "Hacks, Spooks and Data Encryption," Dr. Dobb's Journal, September 1990, pp. 127–134, 147–149.

Ex.1007 / Page 167 of 1158
APPLE INC.

Viruses in Graphics Files

"Avoid detection and dodge selection."

—What Charles Darwin might have said if asked about the fundamental behavior of computer viruses.

Every computer user worries about computer viruses. These often harmful and sometimes nearly undetectable programs are the subject of much consumer literature and urban folklore.

It is hard to generalize about computer viruses, but they typically are not very complex. In most cases, a virus is just a program that is written to replicate itself, avoid detection by both the computer user and the operating environment, and perhaps do a few other things along the way. Some viruses are meant to cause harm (e.g., destroy files and bring networks to a screeching halt), of course, but others are only inadvertently harmful due to logic errors in their own code or because of incompatibilities between the code and the system on which it is running. Still other viruses are programmed to be merely annoying; in some cases, they taunt their victim, who realizes to his dismay that something alien has at least partial control over his computer.

Why do we use the term *computer virus*? At first, the term may seem to be simply a play on the well-known phrase *software bug*. But there's more to it than that. The operational characteristics of computer viruses bear an amazing similarity to those of biological viruses. The primary goal of a biological virus is to reproduce. A virus is only a fragment of RNA or DNA, and therefore does not constitute what most people consider to be a living organism. Because it cannot reproduce on its own, a biological virus must infect a living host in which to reproduce—a process which may result in the disease or death of the host.

When an infected host program is executed by the operating system, the operating system unknowingly executes the viral code as well. The executed viral code is designed to seek out other compatible host programs and to attach copies of itself to them as well. When the code in a virus infects another program, the virus is said to have reproduced. Most executable programs, such as .COM and .EXE files on MS-DOS systems, contain machine code and are directly executed by an operating system. A virus can only infect files that it is designed to infect.

Some viruses are designed so they can attach themselves to batch files, shell scripts, the boot sectors on hard and floppy disks, and even spreadsheet and word processor macros. If it is code that can be executed by a software program, it is fair game for infection by a virus.

Ex.1007 / Page 168 of 1158
APPLE INC.

What about graphics files? Can they be infected with a virus? Is your computer system in any danger from infected graphics files?

Graphics files are generally collections of data and as such are not executed by a computer's operating system. Programs that use graphics files, such as display and editing programs, simply read the data in a graphics file into memory and then modify it for rendering to an output device. Graphics files that consist of data cannot be infected by a virus because the code is not executed. Static graphics files (i.e., those containing no code) are safe from infection.

Some graphics files, such as those used in multimedia applications, do have the capability of storing instructions that can be executed by specific software applications. Such instructions might display text, create sound, pop up menus, and read data from other files. Object-oriented files containing data and the code necessary to render the data are also in this category. These types of files are theoretically susceptible to virus infection. At this point, however, none to our knowledge have been attacked by viruses. This may change, however, as instruction sequences necessary for the proper rendering of a particular file become more complex. However, none of the file formats in common use support this level of complexity.

NOTE

> What about page description languages (PDLs) such as PostScript and hypertext languages such as HTML? Such languages are not actually graphics file formats, but are instead collections of interpreted statements that may contain or reference graphics data. Although the graphics data itself is not a target for a virus, the interpreted language code can be altered, and known security holes in the programming language can be exploited.

In summary, graphics files are very unlikely candidates for infection by computer viruses. In fact, most virus detection programs will not even bother to scan graphics files unless told explicitly to do so. Of course, a virus could still do nasty things to a graphics file, just as it could to any other type of file—for example, it could copy, move, alter, corrupt, or delete a graphics file, or it could append data to the file to cause it to grow in size. But a computer virus cannot use a data-only graphics file—or any data file, for that matter—to reproduce.

Ex.1007 / Page 169 of 1158
APPLE INC.

Designing Your Own Format

We find it hard to imagine why anyone would think that the world needs another graphics file format. And, in fact, we don't want to give the impression that we're encouraging such behavior. But given the fact that people can and will create new formats, we'd like to leave you with some pointers.

Why Even Consider It?

The truth is that this book does not even begin to include all of the hundreds of more obscure formats, some of which are used only privately and remain inside company walls. Companies wishing the output of their products to remain proprietary will always find a way to make it so and thus will continue to develop new formats.

Designing your own format also will help you avoid trouble should the use of someone else's format one day be restricted through legal action. The use of the GIF file format has recently come under licensing restrictions requiring that a royality fee be paid for software that reads or writes the GIF file format. Payment of this fee has been actively enforced through the threat of legal action both by both the owners of the GIF format and the owners of the Lempel-Ziv-Welch (LZW) compression algorithm used by GIF. Remember that even though many formats appear to be freely and publicly available, very few actually are.

Of course there are functional reasons for designing your own format. You may decide that an appropriate format doesn't yet exist, for instance, and thus feel compelled to create a new one. Reasoning leading to this decision is always suspect, however, and sending yet another format out into the world might even decrease your market share in this era of increasing interoperability and file sharing. The unfortunate reality is that file formats are usually created to support applications after the fact. In the modern world, marketing decisions and speculation about the future evolution of the supporting operating system and hardware platform play large parts in the development of program specifications from the very start. So we urge you to consider designing your application around a set of existing formats, or at least a format model.

But If You Must . . .

With that said, consider the following guidelines if you persist in designing your own:

- Study everybody else's mistakes. No matter what you think, you're not smart enough to avoid them all, unless you see them first.

Ex.1007 / Page 170 of 1158
APPLE INC.

- Plan for future revisions. Give plenty of room for future expansion of, and changes to, data. Avoid building limitations into your format that will one day force you to make kludges to support further revisions.
- Keep it simple. The last thing the world needs is another "write-only" format. The object is to make it easy to read, not easy to write.
- Document everything! Use consistent terminology that everyone understands and will continue to understand until the end of time. Number your documentation revisions with the format revisions; that way, it will be obvious if you "forget" to document a new feature.
- Write the format before, not after, your application. Build the application around it. Don't make convenience revisions no matter what the provocation.
- Avoid machine dependencies; at the same time, don't add complications designed to support portability.
- Find some unambiguous means by which a reading application can identify the format.
- Write a detailed file format specification and make it available. Do not discourage people who are interested in your format by refusing to supply them with information simply because they are not registered users of your product. The more widely distributed your information, the greater the potential acceptance of your format will be.
- If your format is truly superior, market, market, market! Write software applications that use your format. Formats gain currency almost entirely through the marketing power of companies and their software. If your format is unique in the way it stores information, and you feel that it fills a niche that other formats don't, then advertise that fact.
- Explicitly place the format in the public domain. If that is too threatening in the context of your company model, allow use with attribution. Do not discourage the spread of your format by including threats about copyright infringement and proprietary information in the specification. This only prevents your format from becoming widely accepted, and it alienates programmers who would otherwise be happy to further your company's plan of world domination for free.
- Develop canonical test files and make them freely available. Mark them as such *in the image*, with your company name, the format type, color model, resolution, and pixel depth at the very least. They will be a good form of advertising for you and your company and are sure to be widely distributed with no effort on your part.

Ex.1007 / Page 171 of 1158
APPLE INC.

One Last Word

Remember that a lot of code is already out there and plenty of libraries are available in source form that may be able to supply your needs. Consider this statement from the FAQ (Frequently Asked Questions list) from the *comp.graphics.misc* newsgroup on the Internet:

> Format documents for TIFF, IFF, BIFF, NFF, OFF, FITS, etc. You almost certainly don't need these. Read the section on free image manipulation software. Get one or more of these packages and look through them. Chances are excellent that the image converter you were going to write is already there.

Writing a File Format Specification

If you are going to write a graphics file format specification, the first thing to do is to make a list of the types of information you are going to store. Then make a second list, of all the auxiliary data a program will need in order to render the data you wrote down in the first list. Get the picture? In compiling the second list, you'll almost certainly find that you forgot something in the first—and vice versa.

The next thing to do is to look at all the format specifications like yours and notice where the writers went astray. Now go back and fix up your list. Now iterate. This is an exercise in honesty, intelligence, and diligence. Nobody—and we mean nobody—has gotten it right yet. Maybe you'll be the first.

Suggestions for Writing Specs

When you read through your pile of format specifications you'll find out that no two are alike (unless they are written by the government or military—in that case they are *all* alike). And you will discover that most are poorly written and quite complex as well. How can you avoid making the same mistake? Here are a few suggestions.

- A large spec needs a table of contents, a bibliography, and an index. However, because most specs are usually no more than a dozen pages or so, most will not need to have this type of organization.
- On the spec's cover sheet, give full information about your company, any products associated with the format, the format version, date of release, where the latest copy of the spec may be obtained, and how developers may get in contact with you to ask questions.

Ex.1007 / Page 172 of 1158
APPLE INC.

- Detail the full history of the spec, including the differences between the current version and all previous versions, and not just the dates of its revision. Tell why the format was created. Detail some insights of how it was designed. Speculate on what features future versions might contain. And give the names of your developers and other people involved. Show the human thought that exists behind the cold chunk of data that is your format.
- List the features of your format and explain how you intend that it should be used and not used. Give a rationale for excluding features that people might find desirable (e.g., supporting multiple images, popular methods of data compression, and so on). Give the developer reasons why she should use your format (and why she should not bother with others).
- Include both block diagrams and ANSI C code examples of the format's internal data structures. Illustrate actual examples of ASCII file format data and hexadecimal dumps of binary format data. Such information helps programmers quickly and correctly implement code that uses your format.
- If your format includes one or more forms of data compression, error checking, encryption, etc., place this information in a separate section, and give plenty of examples (both written and code) of how these algorithms work. Include mathematical formulas if you believe that they will make the concepts clearer.
- Make the specification available in both hardcopy and electronic form. The hardcopy version should be formatted as a technical document using a font that won't degrade badly when the spec is photocopied or faxed. Use a standard sized page layout so the spec isn't a hassle to fit in an envelope when mailed. The electronic version should be available as both ASCII text and PostScript files. Making the spec available in a word processing format (such as Microsoft Word or FrameMaker) is nice but not absolutely necessary.
- Consider making a developer's toolkit for your format. A collection of benchmark graphics files (one of each flavor of your format) and a parser written in ANSI C that reads and writes your format will be of tremendous help to programmers. Such a kit will allow developers to implement your format quickly in their products and will help minimize the chances of numerous software packages appearing that create graphics files that don't meet your spec. Examples of formats with toolkits include TIFF, TGA, WPG, and PNG.

Ex.1007 / Page 173 of 1158
APPLE INC.

- Submit your specification to every FTP, Gopher, and Web site, and to every BBS that archives file format specs. Notify the maintainers of related FAQs (graphics, animation, multimedia, audio, medical, etc.) that your format exists, and ask for a mention. Send your literature to graphics and imaging software companies to sell support of your format and/or software products.

Examples of some well-written format specs that we've encountered include TGA, TIFF, PNG, EPSF, and PostScript.

We've also found that some specs are well-written, but contain so much extraneous information that they are overly complex and too tedious to read. Most government and military formats are in this group, as are those created by committee; examples in this category are CALS, NITF, NAPLPS, IGES, GKS, and CGM.

And finally, format specs such as PCX, GIF, JFIF, and Sun Raster definitely fall into the "don't let this happen to you" category.

Suggestions for Good Technical Writing

Here are a few, more general, guidelines on good technical writing:

- Write in a tutorial style with explanations and examples of your topics. Don't just give a terse, dictionary description of a topic that will leave readers confused and in need of more information.
- Write in simple terms. Don't assume that your readers enjoy 70-word sentences or that they have advanced degrees in mathematics or computer graphics.
- Have other people read and attempt to understand your spec. Don't assume that just because you understand what you've written, every other reader will as well. You, as the file format specification's author, understand the format inside and out. Your readers, however, do not. Your job is to bring them up to speed. Don't be embarrassed if you don't know how to do this; good technical writing is a highly specialized skill. An explanation that seems clear to you may be just gibberish to your readers. Get help with your writing if you need it.
- Write for a world-wide audience of programmers. Omit slang or regional expressions that a developer living on the other side of the planet might not understand.

Ex.1007 / Page 174 of 1158
APPLE INC.

- Remember that programs that check spelling and grammar are our friends. Use them.

Trademarks, Patents, and Copyrights

You probably think of a trademark, whether it's on a Microsoft Windows logo, a Xerox machine, or a box of Kleenex, as a mark of product identification. You may think of a patent as an ownership claim on a strange invention or chemical process. And when you think of a copyright, you may visualize a book, a magazine article, or a piece of music branded with a name and date claiming ownership of that work.

Where do graphics file formats fit into this scheme? Let's start to answer this question by examining trademarks, then patents, and finally copyrights.

NOTE

> The authors are not attorneys, and this information should not be construed as legal advice. We advise you to contact an intellectual property attorney if you have a need for absolutely accurate and up-to-date information.

Trademarks

Trademarks are words, names, or symbols used to identify a product or a service. Owning a trademark doesn't mean that you own the actual words associated with the trademark claim (although Eastman Kodak and Xerox Corporation might disagree). The trademark only bestows ownership of a word or phrase within a certain context, such as the word *windows* within the context of the computer industry.

Trademarks are important in the world of graphics file formats. CompuServe owns the trademark "GIF"; Adobe (and formerly Aldus) owns the trademark "TIFF"; and Microsoft owns the trademark "RIFF". You get the idea.

Graphical logos can also be the subject of trademarks. The Apple Computer and Microsoft Windows logos are registered trademarks belonging to those corporations. Original artwork used as a trademark can, unlike words, be owned by the holder of the trademark. An original piece of artwork is considered to be a "work of authorship," while a few words of the English (or other) language strung together in a clever way is not.

In summary, you can't trademark a graphics file or its format, but you can certainly trademark its name and logo.

Ex.1007 / Page 175 of 1158
APPLE INC.

Patents

What about patents? Can a graphics file format, the contents of such a file, or the file format itself be patented?

A patent is a claim to the ownership of an invention, process, or design that is "new, useful, and nonobvious." Inventions that are mechanical, chemical, electrical, or electronic in nature are covered by "utility patents." Functional and artistic designs are covered by "design patents." Which patent would apply to a file format?

Remember that file and data formats are just specific arrangements of information present in some type of medium or media. They are not the information or the media itself.

A utility patent would apply to the invention of storing information on a hard disk device, or text on the pages of a book, or even the concept of the book itself. But it would not apply to the format of the information, unless the format were a necessary part of the invention itself.

A file format is more of a functional design, as is the shape of an airplane or a cathode ray tube, rather than an artistic design, such as the pattern on a piece of fabric. Does this mean that you can create a new graphics file format and have its design patented?

To be awarded a patent, inventions and designs must be new, useful, and not derived from a pre-existing work. Although file formats exist in many different forms, they all still do the same job: store graphical data in an analog or digital medium. It is likely that attempting to obtain a design patent for a new file format would be turned down based on the existence of prior art.

Copyrights

Can a graphics file be copyrighted?

A graphics file itself cannot normally be copyrighted under United States copyright laws (although the rulings of some judges may disagree). The specification of a format and the "contents" of a graphics file, however, are subject to copyright. In other words, your secret barbecue sauce, or its recipe, can win a blue ribbon at the county fair but not the jar you put the sauce in, or the paper you wrote the recipe on.

For anything to be copyrighted it must be:

- A work of authorship

Ex.1007 / Page 176 of 1158
APPLE INC.

- Fixed in a tangible medium of expression

The description of a graphics format does meet both of these criteria if it is both fixed in a medium (printed on paper or stored on disk) and a work of authorship (not copied from a pre-existing work). Any file format specification that meets these two requirements is protected under the copyright laws.

A graphics file created using a format description, however, meets the second criteria but not the first. That is, the file itself is not considered to be a work of authorship. The file itself is considered instead to be an idea or a system and is therefore not protected by the laws of copyright.

So the description of a file format is copyrightable, but the format as it exists in its medium is not. What about the graphics data that the file contains?

If the graphics data written to a graphics file also meets the above two criteria, it is also protected by the copyright laws as intellectual property. You will not waive your copyright protection by storing any original information using a graphics file format.

Explicit versus implicit copyrights

How do you copyright the contents of a graphics file or a file format specification? There are several levels of copyright: formal, explicit, and implicit.

You can formally register the copyright of your work to establish priority as the creator of the work. This action gives you extra protection if you intend to sell or otherwise assign your copyright, or if you need to defend yourself in the event that the ownership of your work is disputed in a court of law. For the most up-to-date information about copyright registration, consult the U.S. Copyright Office (or the appropriate office in your own country) or an intellectual property attorney.

Most people copyright their work explicitly, but informally, by attaching a copyright notice to the work. (We'll describe how you do this in the sections that follow.) However, neither formal registration nor even the attachment of an explicit copyright notice is needed to establish copyright. Thanks to the Berne Convention on copyrights, the contents of any graphics files created after March 1, 1989, are automatically and implicitly copyrighted and protected, regardless of whether a copyright notice is actually present in the file. In fact, even all the posts on USENET and all email sent across the Internet are automatically copyrighted by these international laws.

Unfortunately, many people do not realize that even though they don't see an explicit copyright notice on a file, the information in that file is still subject to

Ex.1007 / Page 177 of 1158
APPLE INC.

Work for Hire

In many cases a copyright is not automatically assigned, such as with work-for-hire. If you are paid by someone to create a copyrightable work, the copyright belongs to your employer and not to you.

protection under the copyright laws. We recommend that you include a visible copyright notice on your file. Doing so will drive the point home that the contents of your graphics files are at least in some ways protected by copyright laws.

A minimal copyright notice looks like this:

```
Copyright date(s) by author(s)
```

This notice visually establishes the fact the contents of your file are copyrighted on a given year(s) and indicates who holds the copyright. For example:

```
Copyright 1995-96 by James D. Murray
```

You might enhance your copyright notices by stating:

```
Copyright (C) 1995-96 by James D. Murray. All rights reserved.
```

The (C) is an ASCII attempt to represent the "c in a circle" (©) copyright symbol. Note that you must include either the word *Copyright* or the copyright symbol in your statement, It is redundant, but harmless, to include both. Note also that the (C) character that people put in ASCII files has not yet been accepted as a valid copyright symbol by any court of law.

The phrase *All rights reserved* was a requirement of several countries many years ago to consider a copyright notice valid. Under current international copyright laws, this phrase is no longer required, but many people still use it.

Object code copyright

An object code copyright is a human-readable copyright notice that has been embedded into the object code of a library mode or executable file. This notice allows anyone who is performing a string search on an object file to see that the intellectual contents of the file are protected by copyright.

In ANSI C, you could use the following line of code to embed a copyright string into an object file:

```
static const char * const Copyright = "Copyright 1996 by James D. Murray";
```

This code creates a static constant named `Copyright` that contains your copyright string. This is also the string that should be displayed to the user with

Ex.1007 / Page 178 of 1158
APPLE INC.

other information about the program.

Public domain

Not everybody feels the need to copyright their work. Some people explicitly want to share the fruits of their labors. Instead of including a copyright notice in your file, you may choose to place the contents of your file format into the public domain. Doing so allows anybody to do anything they wish with your work. By placing your work in the public domain, you are, in effect, making a "no strings attached" contribution to the freely accessible knowledge base of humankind.

You place the contents of your graphics files into the public domain by including a statement in your work such as:

```
I yournamehere grant this nameofyourworkhere to the public domain.
```

This statement is a legal notice that your work may be freely used and distributed by anyone as he or she sees fit.

License notice

What if you don't want to prevent people from using or distributing your graphics by waving around your copyright, but you don't want to release your work into the public domain only to have some rogues claim your work as theirs? For one possible answer, let's look at the copyright notice on the Graphics File Formats Frequently Asked Questions (FAQ) listing that circulates on USENET:

```
This FAQ is Copyright 1994-96 by James D. Murray. This work may be
reproduced, in whole or in part, using any medium, including, but not
limited to, electronic transmission, CD-ROM, published in print, under the
condition that this copyright notice remains intact.
```

This copyright notice includes a statement, called a license notice, that declares how the author intends the information in the listing to be used. And, as you can read, the author allows anyone to do practically anything he or she wants with the information in this listing, as long as the author is given proper credit as the creator and the maintainer of the listing.

What we have here is a copyright notice establishing ownership of a work which includes a written statement of what the owner of the work considers to be "fair use" of the copyrighted material. In this case, the author has all but placed the original information in the Graphics File Formats FAQ into the public domain; all he asks is that he be recognized for his efforts if the information contained in the FAQ is ever used. What a nice guy!

Ex.1007 / Page 179 of 1158
APPLE INC.

Rights of ownership

Note that you must be the owner of the work to explicitly copyright it or place it into the public domain. Willfully or unknowingly using or distributing a copyrighted work without permission of the copyright holder, for profit or for free, is a clear and prosecutable violation under the copyright laws. Your offense is even worse if your "use" of the material has hurt the commercial value of the property itself.

The owner of a copyright has several exclusive rights to his copyrighted work. Four of these rights that apply directly to the contents of graphics files include:

- Reproduction rights: the right to copy, transcribe, duplicate, or imitate the work in a fixed form.
- Modification rights: the right to derive a new work based on the copyrighted work.
- Distribution rights: the right to distribute the work by sale, rental, leasing, or lending.
- Public display rights: the right to show or transmit the work to the public by direct or indirect means.

Can you get into trouble distributing graphics files that you do not own? You bet you can. Violating the exclusive rights of a copyright owner will earn you the exclusive title of "copyright infringer" by most countries on Earth.

Let's say you come across a nice photograph or piece of artwork in a magazine or on videotape, and you decide to scan or capture it, save it to disk using a popular graphics file format, trim some of the image away and maybe add a border, and upload it to a BBS or post it on USENET or another information service. By doing so you have infringed upon the copyright holder's reproduction rights (scanning or capturing the work and saving it), modification rights (altering the work using a graphical editor), and distribution rights (uploading the file and its reproduced work).

A court of law may award actual and statutory damages for such infringements, potentially ranging from tens of thousands to millions of dollars.

To stay out of trouble, you must assume that you do not have permission to copy, modify, or distribute any graphics files unless you have explicit permission to do so from the owners of the file's contents. Scanning a page from a magazine and storing the image in a file does not give you ownership of the image, only ownership of the file. The file's contents still belong to the holder of the copyright of the text, photograph, or artwork that you scanned.

Remember, the ownership of the graphics file itself is not the issue. You must always be aware of who owns and has the rights to the graphics file's contents.

Ex.1007 / Page 180 of 1158
APPLE INC.

Copyright notices in graphics files

By now you are probably wondering how you can include a copyright notice in a graphics file. Some file formats have a text field, often in the header, specifically reserved for a copyright notice; for example, TIFF, SPIFF, DPX, and PNG support such a field. Many formats support the storage of text comments in a user-defined data field; for example, GIF, IFF, and TGA allow text comments. Such comments are used to describe anything the file writer wishes, which is usually the contents of a file, the name of the author, and the copyright notice.

In formats that lack comment or copyright fields (e.g., PCX and BMP), copyright notices can find their way into areas of the file reserved for future expansion of the file format, or they can simply be tacked onto the end of the file itself. Neither of these methods is recommended, because they will surely cause some file format readers to report an error—or simply blow up—when they read such copyright-kludged graphics files.

In you can't use a format that properly supports the storage of a copyright notice, you should include an external file bearing your copyright notice and terms of usage and listing the names and descriptions of all of the copyrighted graphics files that you are distributing. This method has the advantage of making your copyright notice human-readable, which is helpful because very few graphics file viewers are able to display text comments embedded within a graphics file.

Summary

To summarize the facts of trademarks, patents, and copyrights applied to graphics files, remember that a file format specification, and the contents of a graphics file, can be protected by a copyright, but the graphics file itself cannot. And, until the federal courts rule otherwise, file formats—or more precisely, their contents—may be subject to copyright but not to trademarks or patents.

For Further Information on Copyrights and Patents

For more information on copyright, please refer to the Copyright FAQs found on the *misc.legal*, *misc.legal.computing*, *misc.int-property*, and *comp.patents* newsgroups, as well as the FTP sites:

ftp://rtfm.mit.edu/pub/usenet/news.answers/law/Copyright-FAQ/
ftp://rtfm.mit.edu/pub/usenet/news.answers/law/Copyright-FAQ/myths/

Quite a few copyright discussions also occur on the *comp.infosystems.www.** newsgroups.

Ex.1007 / Page 181 of 1158
APPLE INC.

For information about the Internet Patent News Service mailing list, send email to *patents@world.std.com.*

Information on patents, copyrights, and intellectual property may also be found at:

http://www.questel.orbit.com/patents/readings.html
Patent, trademark, scientific, chemical, business, and news information

http://www.uspto.gov
U.S. Patent and Trademark Office

http://www.spi.org
Software Patent Institute

ftp://comments.uspto.gov/pub/software_hearings
Transcriptions of hearing on software patents

ftp://marvel.loc.gov/pub/copyright
Forms and information from the U.S. Copyright Office

http://www.eff.org/pub/CAF/law/ip-primer
Intellectual property law primer for multimedia developers

http://www.eff.org/pub/CAF/law/multimedia-copyright
http://www.eff.org/pub/CAF/law/multimedia-handbook
Copyright information for multimedia developers

Ex.1007 / Page 182 of 1158
APPLE INC.

CHAPTER 9

Data Compression

Compression is the process used to reduce the physical size of a block of information. In computer graphics, we're interested in reducing the size of a block of graphics data so we can fit more information in a given physical storage space. We also might use compression to fit larger images in a block of memory of a given size. You may find when you examine a particular file format specification that the term *data encoding* is used to refer to algorithms that perform compression. Data encoding is actually a broader term than data compression. Data compression is a type of data encoding, and one that is used to reduce the size of data. Other types of data encoding are involved in encryption (cryptography) and data transmission (e.g., Morse code).

A *compressor*, naturally enough, performs compression, and a *decompressor* reconstructs the original data. Although this may seem obvious, a decompressor can operate only by using knowledge of the compression algorithm used to convert the original data into its compressed form. What this means in practice is that there is no way for you to avoid understanding the compression algorithms in the market today if you are interested in manipulating data files. At a minimum, you need general knowledge of the conceptual basis of the algorithms.

If you read through a number of specification documents, you'll find that most formats incorporate some kind of compression method, no matter how rudimentary. You'll also find that only a few different compression schemes are in common use throughout the industry. The most common of these schemes are variants of the following methods, which we discuss in the sections below.

- Run-length encoding (RLE)
- Lempel-Ziv-Welch (LZW)

Ex.1007 / Page 183 of 1158
APPLE INC.

- CCITT (one type of CCITT compression is a variant on Huffman encoding)
- Discrete Cosine Transform (DCT) (used in the JPEG compression we discuss in this chapter)
- JBIG
- ART
- Fractal

In addition, we discuss pixel packing, which is not a compression method per se but an efficient way to store data in contiguous bytes of memory.

Data compression works somewhat differently for the three common types of graphics data: bitmap, vector, and metafile. In bitmap files, *only* the image data is usually compressed; the header and any other data (color map, footer, and so on) are always left uncompressed. This uncompressed data makes up only a small portion of a typical bitmap file.

Vector files normally do not incorporate a native form of data compression. Vector files store a mathematical description of an image rather than the image data itself. There are reasons why vector files are rarely compressed:

- The expression of the image data is already compact in design, so data compression would have little effect.
- Vector images are typically fast to read but slow to reconstruct; adding the overhead of decompression would make rendering them still slower.

If a vector file is compressed at all, you will usually find the entire file compressed, header and all.

In metafiles, data compression schemes often resemble those used to compress bitmap files, depending upon the type of data the metafiles contain.

Programmers commonly confuse compression algorithms with the files in which they're used. Many programmers ask vendors or newsgroups for specifications for the CCITT or JPEG file formats. There are no such specifications. Compression algorithms define only how data is encoded, not how it is stored on disk. For detailed information on how data is stored on disk, look to an actual image file format specification, such as BMP or GIF, which will define file headers, byte order, and other issues not covered by discussions of compression algorithms. More complex formats, such as TIFF, may incorporate several different compression algorithms.

The following sections introduce the terms used in discussions of data compression and each of the main types of data compression algorithms used for graphics file formats today.

Ex.1007 / Page 184 of 1158
APPLE INC.

Data Compression Terminology

This section describes the terms you'll encounter when you read about data compression schemes in this chapter and in graphics file format specifications.

The terms *unencoded data* and *raw data* describe data before it has been compressed, and the terms *encoded data* and *compressed data* describe the same information after it has been compressed.

The term *compression ratio* is used to refer to the ratio of uncompressed data to compressed data. Thus, a 10:1 compression ratio is considered five times more efficient than 2:1. Of course, data compressed using an algorithm yielding 10:1 compression is five times smaller than the same data compressed using an algorithm yielding 2:1 compression. In practice, because only image data is normally compressed, analysis of compression ratios provided by various algorithms must take into account the absolute sizes of the files tested.

Physical and Logical Compression

Compression algorithms are often described as squeezing, squashing, crunching, or imploding data, but these are not very accurate descriptions of what is actually happening. Although the major use of compression is to make data use less disk space, compression does not actually physically cram the data into a smaller size package in any meaningful sense.

Instead, compression algorithms are used to re-encode data into a different, more compact representation conveying the same information. In other words, fewer words are used to convey the same meaning, without actually saying the same thing.

The distinction between *physical* and *logical compression* methods is made on the basis of how the data is compressed or, more precisely, how the data is rearranged into a more compact form. Physical compression is performed on data exclusive of the information it contains; it only translates a series of bits from one pattern to another, more compact one. While the resulting compressed data may be related to the original data in a mechanical way, that relationship will not be obvious to us.

Physical compression methods typically produce strings of gibberish, at least relative to the information content of the original data. The resulting block of compressed data is normally smaller than the original because the physical compression algorithm has removed the redundancy that existed in the data itself. Most of the compression methods discussed in this chapter are physical methods.

Ex.1007 / Page 185 of 1158
APPLE INC.

Logical compression is accomplished through the process of logical substitution—that is, replacing one alphabetic, numeric, or binary symbol with another. Changing "United States of America" to "USA" is a good example of logical substitution, because "USA" is derived directly from the information contained in the string "United States of America" and retains some of its meaning. In a similar fashion "can't" can be logically substituted for "cannot". Logical compression works only on data at the character level or higher and is based exclusively on information contained within the data. Logical compression is generally not used in image data compression.

Symmetric and Asymmetric Compression

Compression algorithms can also be divided into two categories: *symmetric* and *asymmetric.* A symmetric compression method uses roughly the same algorithms, and performs the same amount of work, for compression as it does for decompression. For example, a data transmission application where compression and decompression are both being done on the fly will usually require a symmetric algorithm for the greatest efficiency.

Asymmetric methods require substantially more work to go in one direction than they require in the other. Usually, the compression step takes far more time and system resources than the decompression step. In the real world this makes sense. For example, if we are making an image database in which an image will be compressed once for storage, but decompressed many times for viewing, then we can probably tolerate a much longer time for compression than for decompression. An asymmetric algorithm that uses much CPU time for compression, but is quick to decode, would work well in this case.

Algorithms that are asymmetric in the other direction are less common but have some applications. In making routine backup files, for example, we fully expect that many of the backup files will never be read. A fast compression algorithm that is expensive to decompress might be useful in this case.

Adaptive, Semi-Adaptive, and Non-Adaptive Encoding

Certain dictionary-based encoders, such as CCITT compression algorithms (see the section later in this chapter called "CCITT (Huffman) Encoding") are designed to compress only specific types of data. These *non-adaptive encoders* contain a static dictionary of predefined substrings that are known to occur with high frequency in the data to be encoded. A non-adaptive encoder designed specifically to compress English language text would contain a dictionary with predefined substrings such as "and", "but", "of", and "the", because these substrings appear very frequently in English text.

Ex.1007 / Page 186 of 1158
APPLE INC.

An *adaptive encoder*, on the other hand, carries no preconceived heuristics about the data it is to compress. Adaptive compressors, such as LZW, achieve data independence by building their dictionaries completely from scratch. They do not have a predefined list of static substrings and instead build phrases dynamically as they encode.

Adaptive compression is capable of adjusting to any type of data input and of returning output using the best possible compression ratio. This is in contrast to non-adaptive compressors, which are capable of efficiently encoding only a very select type of input data for which they are designed.

A mixture of these two dictionary encoding methods is the *semi-adaptive encoding method.* A *semi-adaptive encoder* makes an initial pass over the data to build the dictionary and a second pass to perform the actual encoding. Using this method, an optimal dictionary is constructed before any encoding is actually performed.

Lossy and Lossless Compression

The majority of compression schemes we deal with in this chapter are called *lossless.* What does this mean? When a chunk of data is compressed and then decompressed, the original information contained in the data is preserved. No data has been lost or discarded; the data has not been changed in any way.

Lossy compression methods, however, throw away some of the data in an image in order to achieve compression ratios better than that of most lossless compression methods. Some methods contain elaborate heuristic algorithms that adjust themselves to give the maximum amount of compression while changing as little of the visible detail of the image as possible. Other less elegant algorithms might simply discard a least significant portion of each pixel, and, in terms of image quality, hope for the best.

The terms lossy and lossless are sometimes erroneously used to describe the quality of a compressed image. Some people assume that if any image data is lost, this could only degrade the image. The assumption is that we would never want to lose any data at all. This is certainly true if our data consists of text or numerical data that is associated with a file, such as a spreadsheet or a chapter of our great American novel. In graphics applications, however, under certain circumstances data loss may be acceptable, and even recommended.

In practice, a small change in the value of a pixel may well be invisible, especially in high-resolution images where a single pixel is barely visible anyway. Images containing 256 or more colors may have selective pixel values changed with no noticeable effect on the image.

Ex.1007 / Page 187 of 1158
APPLE INC.

In black-and-white images, however, there is obviously no such thing as a small change in a pixel's value: each pixel can only be black or white. Even in black-and-white images, however, if the change simply moves the boundary between a black and a white region by one pixel, the change may be difficult to see and therefore acceptable.

As mentioned in Chapter 2, *Computer Graphics Basics*, the human eye is limited in the number of colors it can distinguish simultaneously, particularly if those colors are not immediately adjacent in the image or are sharply contrasting. An intelligent compression algorithm can take advantage of these limitations, analyze an image on this basis, and achieve significant data size reductions based on the removal of color information not easily noticed by most people.

In case this sounds too much like magic, rest assured that much effort has gone into the development of so-called lossy compression schemes in recent years, and many of these algorithms can achieve substantial compression ratios while retaining good quality images. This is an active field of research, and we are likely to see further developments as our knowledge of the human visual system evolves, and as results from the commercial markets regarding acceptance of lossy images make their way back to academia.

For more information about lossy compression, see the JPEG section later in this chapter.

Pixel Packing

Pixel packing is not so much a method of data compression as it is an efficient way to store data in contiguous bytes of memory. Most bitmap formats use pixel packing to conserve the amount of memory or disk space required to store a bitmap. If you are working with image data that contains four bits per pixel, you might find it convenient to store each pixel in a byte of memory, because a byte is typically the smallest addressable area of memory on most computer systems. You would quickly notice, however, that by using this arrangement, half of each byte is not being used by the pixel data (shown in Figure 9-1, a). Image data containing 4096 4-bit pixels will require 4096 bytes of memory for storage, half of which is wasted.

To save memory, you could resort to pixel packing; instead of storing one 4-bit pixel per byte, you could store two 4-bit pixels per byte (shown in Figure 9-1, b). The size of memory required to hold the 4-bit, 4096-pixel image drops from 4096 bytes to 2048 bytes, only half the memory that was required before.

Pixel packing may seem like common sense, but it is not without cost. Memory-based display hardware usually organizes image data as an array of bytes, each storing one pixel or less. If this is the case, it will actually be faster to store only

Ex.1007 / Page 188 of 1158
APPLE INC.

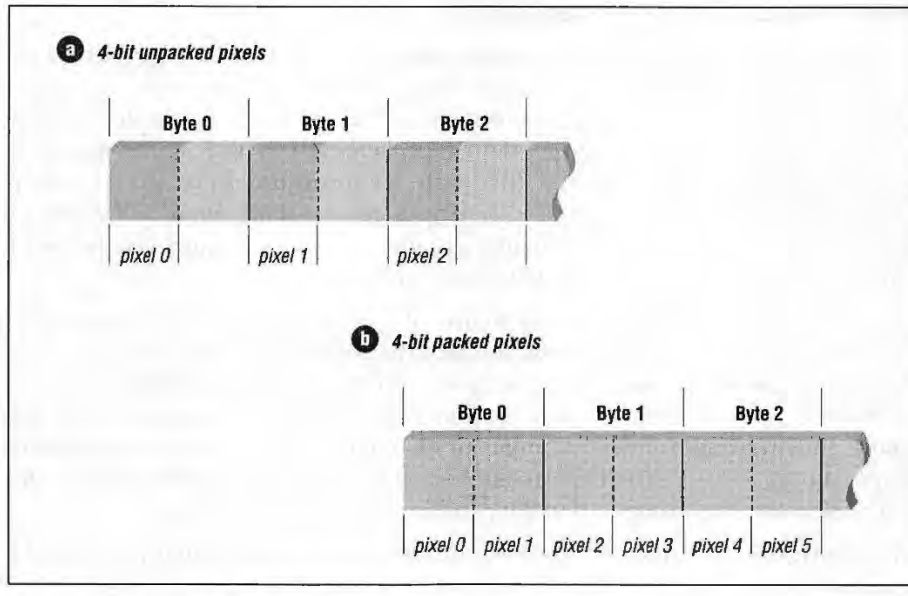


FIGURE 9-1: *Pixel packing*

one 4-bit pixel per byte and read this data directly into memory in the proper format rather than to store two 4-bit pixels per byte, which requires masking and shifting each byte of data to extract and write the proper pixel values. The tradeoff is faster read and write times versus reduced size of the image file. This is a good example of one of the costs of data compression.

Compression always has a cost. In this case, the cost is in the time it takes to unpack each byte into two 4-bit pixels. Other factors may come into play when decompressing image data: buffers need to be allocated and managed; CPU-intensive operations must be executed and serviced; scan-line bookkeeping must be kept current. If you are writing a file reader, you usually have no choice; you *must* support all compression schemes defined in the format specification. If you are writing a file writer, however, you always need to identify the costs and tradeoffs involved in writing compressed files.

At one time in the history of computing, no decision was necessary; disk space was scarce and expensive, so image files needed to be compressed. Now, however, things are different. Hard disks are relatively inexpensive, and alternate distribution and storage media (CD-ROM for instance) are even more so. More and more applications now write image files uncompressed by default. You need to carefully examine the target market of your application before deciding whether to compress or not.

Ex.1007 / Page 189 of 1158
APPLE INC.

Run-Length Encoding (RLE)

Run-length encoding is a data compression algorithm that is supported by most bitmap file formats, such as TIFF, BMP, and PCX. RLE is suited for compressing any type of data regardless of its information content, but the content of the data will affect the compression ratio achieved by RLE. Although most RLE algorithms cannot achieve the high compression ratios of the more advanced compression methods, RLE is both easy to implement and quick to execute, making it a good alternative to either using a complex compression algorithm or leaving your image data uncompressed.

RLE works by reducing the physical size of a repeating string of characters. This repeating string, called a *run*, is typically encoded into two bytes. The first byte represents the number of characters in the run and is called the *run count*. In practice, an encoded run may contain 1 to 128 or 256 characters; the run count usually contains as the number of characters minus one (a value in the range of 0 to 127 or 255). The second byte is the value of the character in the run, which is in the range of 0 to 255, and is called the *run value*.

Uncompressed, a character run of 15 A characters would normally require 15 bytes to store:

```
AAAAAAAAAAAAAAA
```

The same string after RLE encoding would require only two bytes:

```
15A
```

The 15A code generated to represent the character string is called an *RLE packet*. Here, the first byte, 15, is the run count and contains the number of repetitions. The second byte, A, is the run value and contains the actual repeated value in the run.

A new packet is generated each time the run character changes, or each time the number of characters in the run exceeds the maximum count. Assume that our 15-character string now contains four different character runs:

```
AAAAAAbbbXXXXXt
```

Using run-length encoding this could be compressed into four 2-byte packets:

```
6A3b5X1t
```

Thus, after run-length encoding, the 15-byte string would require only eight bytes of data to represent the string, as opposed to the original 15 bytes. In this case, run-length encoding yielded a compression ratio of almost 2 to 1.

Ex.1007 / Page 190 of 1158
APPLE INC.

Long runs are rare in certain types of data. For example, ASCII plaintext (such as the text on the pages of this book) seldom contains long runs. In the previous example, the last run (containing the character t) was only a single character in length; a 1-character run is still a run. Both a run count and a run value must be written for every 2-character run. To encode a run in RLE requires a minimum of two characters worth of information; therefore, a run of single characters actually takes more space. For the same reasons, data consisting entirely of 2-character runs remains the same size after RLE encoding.

In our example, encoding the single character at the end as two bytes did not noticeably hurt our compression ratio because there were so many long character runs in the rest of the data. But observe how RLE encoding doubles the size of the following 14-character string:

```
Xtmprsqzntwlfb
```

After RLE encoding, this string becomes:

```
1X1t1m1p1r1s1q1z1n1t1w1l1f1b
```

RLE schemes are simple and fast, but their compression efficiency depends on the type of image data being encoded. A black-and-white image that is mostly white, such as the page of a book, will encode very well, due to the large amount of contiguous data that is all the same color. An image with many colors that is very busy in appearance, however, such as a photograph, will not encode very well. This is because the complexity of the image is expressed as a large number of different colors. And because of this complexity there will be relatively few runs of the same color.

Variants on Run-Length Encoding

There are a number of variants of run-length encoding. Image data is normally run-length encoded in a sequential process that treats the image data as a 1D stream, rather than as a 2D map of data. In sequential processing, a bitmap is encoded starting at the upper left corner and proceeding from left to right across each scan line (the X axis) to the bottom right corner of the bitmap (shown in Figure 9-2, a). But alternative RLE schemes can also be written to encode data down the length of a bitmap (the Y axis) along the columns (shown in Figure 9-2, b), to encode a bitmap into 2D tiles (shown in Figure 9-2, c), or even to encode pixels on a diagonal in a zig-zag fashion (shown in Figure 9-2, d). Odd RLE variants such as this last one might be used in highly specialized applications but are usually quite rare.

Another seldom-encountered RLE variant is a lossy run-length encoding algorithm. RLE algorithms are normally lossless in their operation. However, discarding data during the encoding process, usually by zeroing out one or two

Ex.1007 / Page 191 of 1158
APPLE INC.

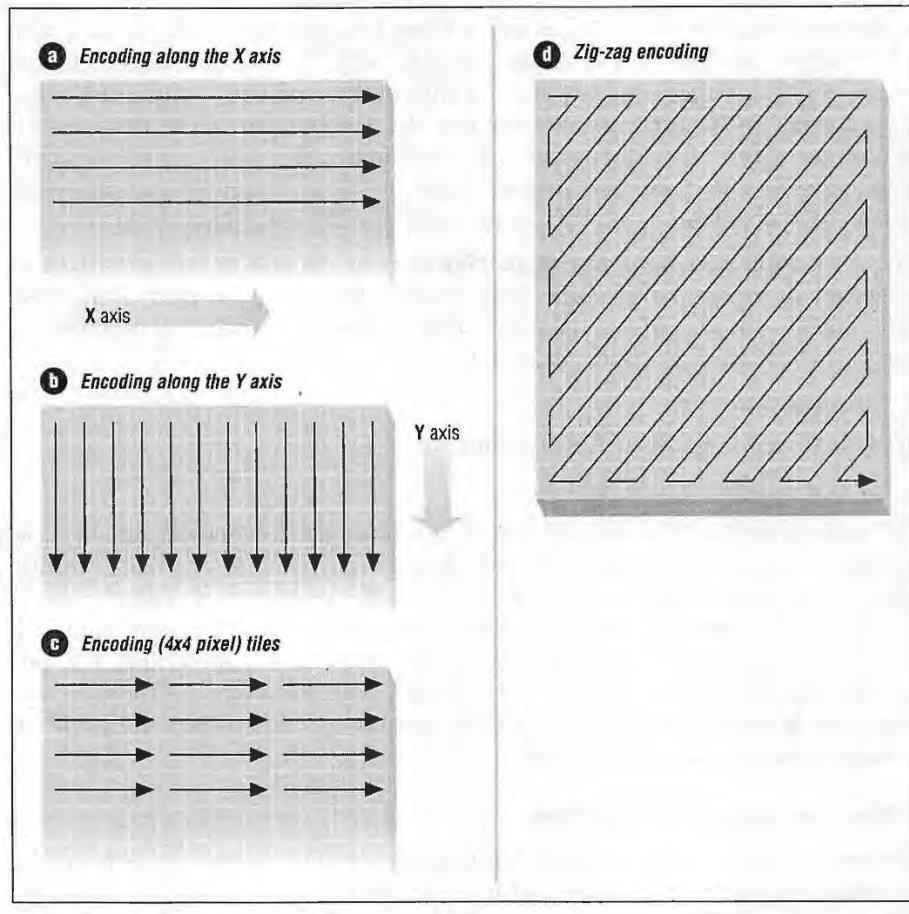


FIGURE 9-2: *Run-length encoding variants*

least significant bits in each pixel, can increase compression ratios without adversely affecting the appearance of very complex images. This RLE variant works well only with real-world images that contain many subtle variations in pixel values.

Make sure that your RLE encoder always stops at the end of each scan line of bitmap data that is being encoded. There are several benefits to doing so. Encoding only a simple scan line at a time means that only a minimal buffer size is required. Encoding only a simple line at a time also prevents a problem known as *cross-coding*.

Ex.1007 / Page 192 of 1158
APPLE INC.

Cross-coding is the merging of scan lines that occurs when the encoded process loses the distinction between the original scan lines. If the data of the individual scan lines is merged by the RLE algorithm, the point where one scan line stopped and another began is lost or, at least, is very hard to detect quickly.

Cross-coding is sometimes done, although we advise against it. It may buy a few extra bytes of data compression, but it complicates the decoding process, adding time cost. For bitmap file formats, this technique defeats the purpose of organizing a bitmap image by scan lines in the first place. Although many file format specifications explicitly state that scan lines should be individually encoded, many applications encode image data as a continuous stream, ignoring scan-line boundaries.

Have you ever encountered an RLE-encoded image file that could be displayed using one application but not using another? Cross-coding is often the the reason. To be safe, decoding and display applications must take cross-coding into account and not assume that an encoded run will always stop at the end of a scan line.

When an encoder is encoding an image, an end-of-scan-line marker is placed in the encoded data to inform the decoding software that the end of the scan line has been reached. This marker is usually a unique packet, explicitly defined in the RLE specification, which cannot be confused with any other data packets. End-of-scan-line markers are usually only one byte in length, so they don't adversely contribute to the size of the encoded data.

Encoding scan lines individually has advantages when an application needs to use only part of an image. Let's say that an image contains 512 scan lines, and we need to display only lines 100 to 110. If we did not know where the scan lines started and ended in the encoded image data, our application would have to decode lines 1 through 100 of the image before finding the ten lines it needed. Of course, if the transitions between scan lines were marked with some sort of easily recognizable delimiting marker, the application could simply read through the encoded data, counting markers until it came to the lines it needed. But this approach would be a rather inefficient one.

Another option for locating the starting point of any particular scan line in a block of encoded data is to construct a scan-line table. A scan-line table usually contains one element for every scan line in the image, and each element holds the offset value of its corresponding scan line. To find the first RLE packet of scan line 10, all a decoder needs to do is seek to the offset position value stored in the tenth element of the scan-line lookup table. A scan-line table could also hold the number of bytes used to encode each scan line. Using this method, to

Ex.1007 / Page 193 of 1158
APPLE INC.

find the first RLE packet of scan line 10, your decoder would add together the values of the first nine elements of the scan-line table. The first packet for scan line 10 would start at this byte offset from the beginning of the RLE-encoded image data.

Bit-, Byte-, and Pixel-Level RLE Schemes

The basic flow of all RLE algorithms is the same, as illustrated in Figure 9-3.

The parts of run-length encoding algorithms that differ are the decisions that are made based on the type of data being decoded (such as the length of data runs). RLE schemes used to encode bitmap graphics are usually divided into classes by the type of atomic (that is, most fundamental) elements that they encode. The three classes used by most graphics file formats are bit-, byte-, and pixel-level RLE.

Bit-level RLE schemes encode runs of multiple bits in a scan line and ignore byte and word boundaries. Only monochrome (black and white), 1-bit images contain a sufficient number of bit runs to make this class of RLE encoding efficient. A typical bit-level RLE scheme encodes runs of one to 128 bits in length in a single-byte packet. The seven least significant bits contain the run count minus one, and the most significant bit contains the value of the bit run, either 0 or 1 (shown in Figure 9-4, a). A run longer than 128 pixels is split across several RLE-encoded packets.

Byte-level RLE schemes encode runs of identical byte values, ignoring individual bits and word boundaries within a scan line. The most common byte-level RLE scheme encodes runs of bytes into 2-byte packets. The first byte contains the run count of 0 to 255, and the second byte contains the value of the byte run. It is also common to supplement the 2-byte encoding scheme with the ability to store literal, unencoded runs of bytes within the encoded data stream as well.

In such a scheme, the seven least significant bits of the first byte hold the run count minus one, and the most significant bit of the first byte is the indicator of the type of run that follows the run count byte (shown in Figure 9-4, b). If the most significant bit is set to 1, it denotes an encoded run (shown in Figure 9-4, c). Encoded runs are decoded by reading the run value and repeating it the number of times indicated by the run count. If the most significant bit is set to 0, a *literal run* is indicated, meaning that the next run count bytes are read literally from the encoded image data (shown in Figure 9-4, d). The run count byte then holds a value in the range of 0 to 127 (the run count minus one). Byte-level RLE schemes are good for image data that is stored as one byte per pixel.

Ex.1007 / Page 194 of 1158
APPLE INC.

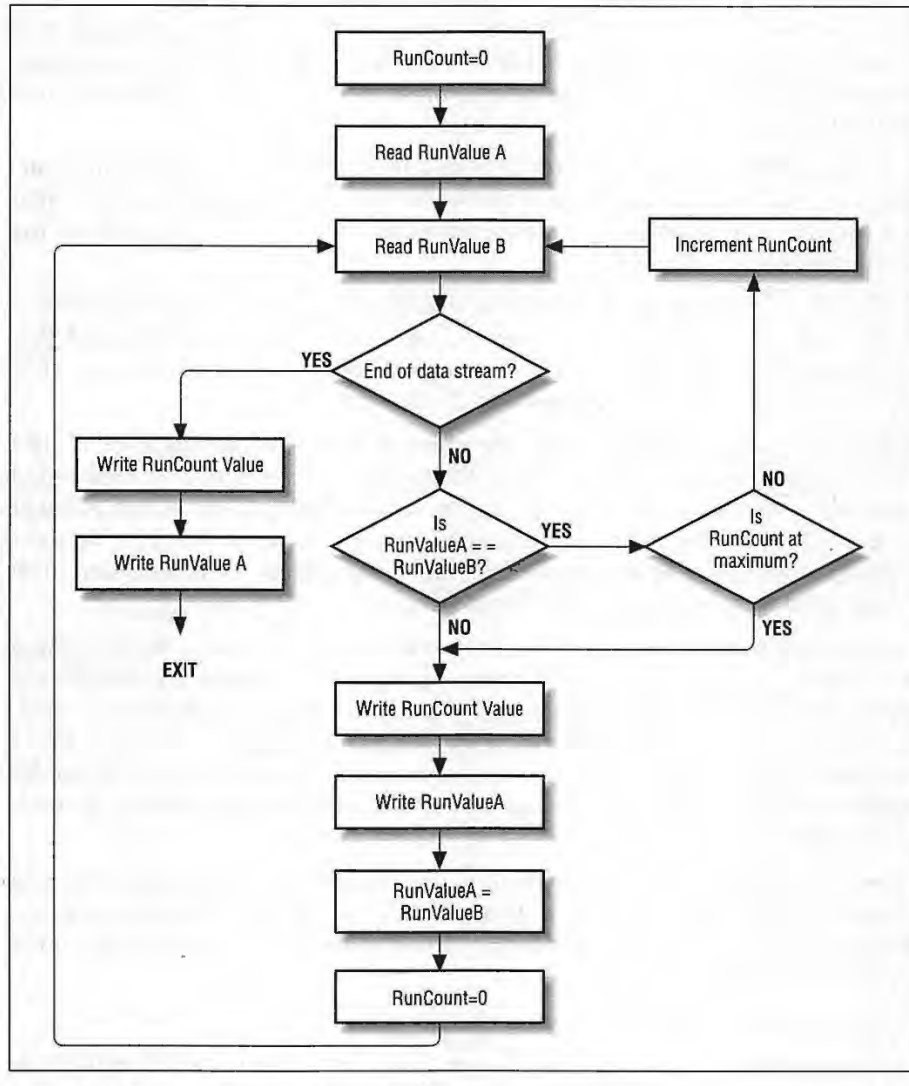


FIGURE 9-3: *Basic run-length encoding flow*

Pixel-level RLE schemes are used when two or more consecutive bytes of image data are used to store single pixel values. At the pixel level, bits are ignored, and bytes are counted only to identify each pixel value. Encoded packet sizes

Ex.1007 / Page 195 of 1158
APPLE INC.

vary depending upon the size of the pixel values being encoded. The number of bits or bytes per pixel is stored in the image file header. A run of image data stored as 3-byte pixel values encodes to a 4-byte packet, with one run-count byte followed by three run-value bytes (shown in Figure 9-4, e). The encoding method remains the same as with the byte-oriented RLE.

It is also possible to employ a literal pixel run encoding by using the most significant bit of the run count as in the byte-level RLE scheme. Remember that the run count in pixel-level RLE schemes is the number of pixels and not the number of bytes in the run.

Earlier in this section, we examined a situation where the string "Xtmprsqzn-twlfb" actually doubled in size when compressed using a conventional RLE method. Each 1-character run in the string became two characters in size. How can we avoid this *negative compression* and still use RLE?

Normally, an RLE method must somehow analyze the uncompressed data stream to determine whether to use a literal pixel run. A stream of data would need to contain many 1- and 2-pixel runs to make using a literal run efficient by encoding all the runs into a single packet. However, there is another method that allows literal runs of pixels to be added to an encoded data stream without being encapsulated into packets.

Consider an RLE scheme that uses three bytes, rather than two, to represent a run (shown in Figure 9-5). The first byte is a *flag value* indicating that the following two bytes are part of an encoded packet. The second byte is the *count value*, and the third byte is the *run value*. When encoding, if a 1-, 2-, or 3-byte character run is encountered, the character values are written directly to the compressed data stream. Because no additional characters are written, no overhead is incurred.

When decoding, a character is read; if the character is a flag value, the run count and run values are read, expanded, and the resulting run written to the data stream. If the character read is not a flag value, it is written directly to the uncompressed data stream.

There are two potential drawbacks to this method:

- The minimum useful run-length size is increased from three characters to four. This could affect compression efficiency with some types of data.
- If the unencoded data stream contains a character value equal to the flag value, it must be compressed into a 3-byte encoded packet as a run length of one. This prevents erroneous flag values from occurring in the compressed data stream. If many of these flag value characters are present, poor compression will result. The RLE algorithm must therefore use a flag value that rarely occurs in the uncompressed data stream.

Ex.1007 / Page 196 of 1158
APPLE INC.

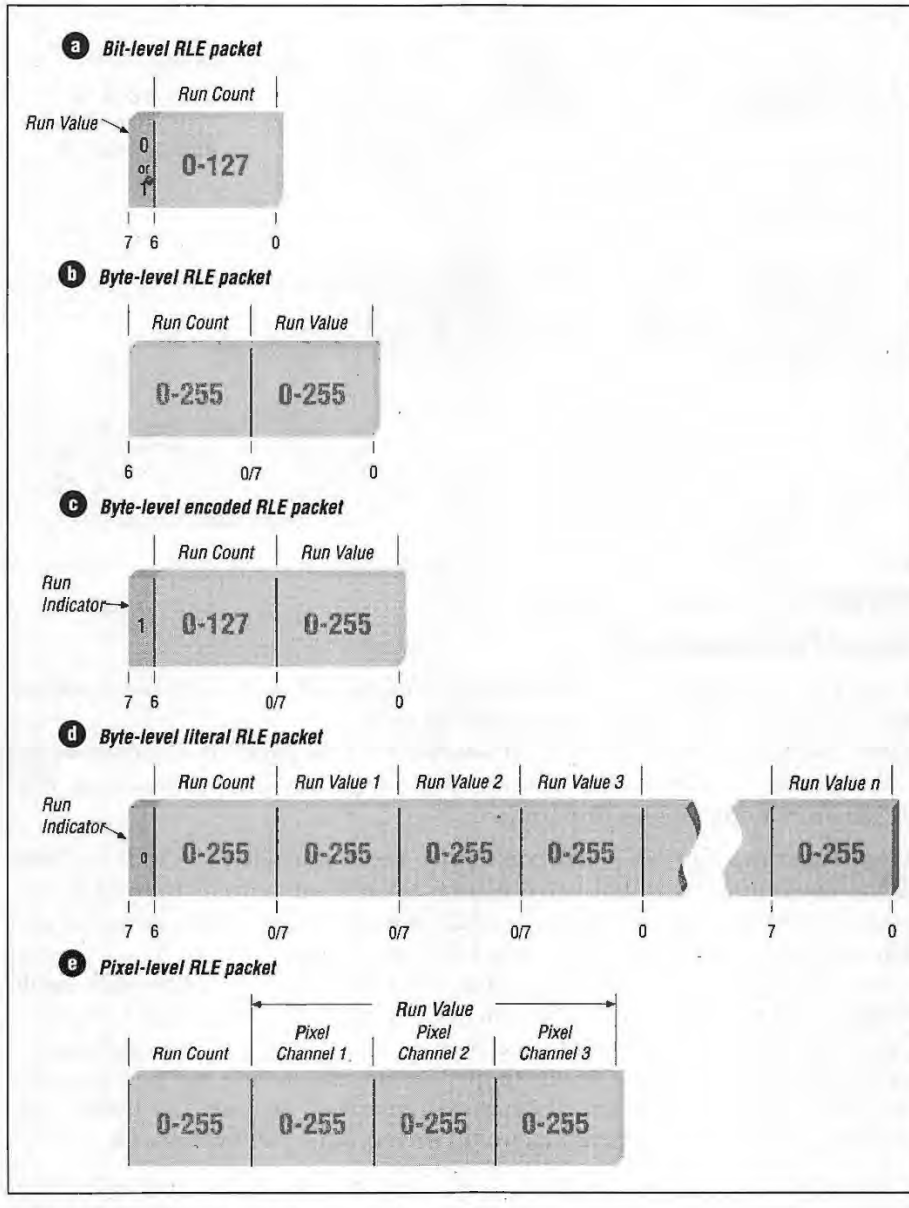


FIGURE 9-4: *Bit-, byte-, and pixel-level RLE schemes*

Ex.1007 / Page 197 of 1158
APPLE INC.

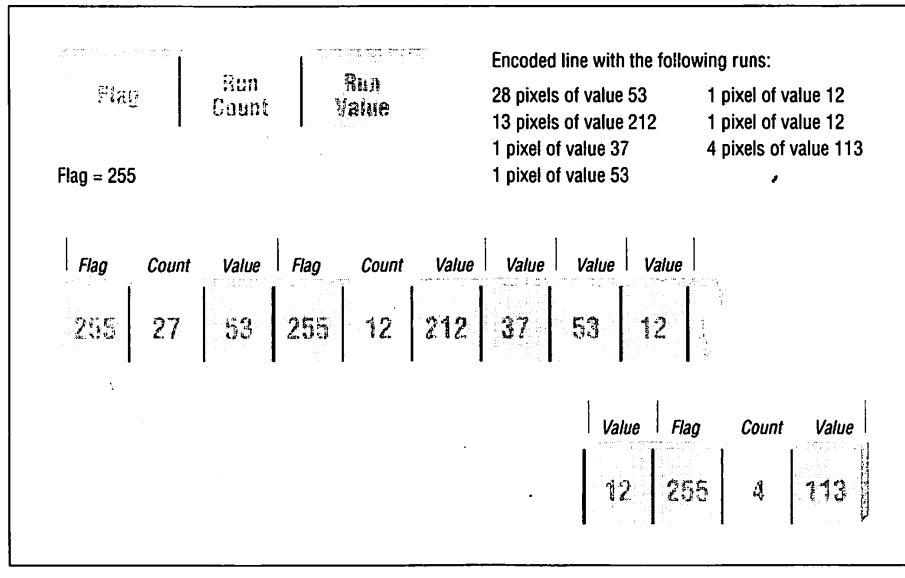


FIGURE 9-5: *RLE scheme with three bytes*

Vertical Replication Packets

Some RLE schemes use other types of encoding packets to increase compression efficiency. One of the most useful of these packets is the *repeat scan line packet,* also known as the *vertical replication packet.* This packet does not store any real scan-line data; instead, it just indicates a repeat of the previous scan line. Here's an example of how this works.

Assume that you have an image containing a scan line 640 bytes wide and that all the pixels in the scan line are the same color. It will require 10 bytes to run-length encode it, assuming that up to 128 bytes can be encoded per packet and that each packet is two bytes in size. Let's also assume that the first 100 scan lines of this image are all the same color. At 10 bytes per scan line, that would produce 1000 bytes of run-length encoded data. If we instead used a vertical replication packet that was only one byte in size (possibly a run-length packet with a run count of 0) we would simply run-length encode the first scan line (10 bytes) and follow it with 99 vertical replication packets (99 bytes). The resulting run-length encoded data would then only be 109 bytes in size.

If the vertical replication packet contains a count byte of the number of scan lines to repeat, we would need only one packet with a count value of 99. The

Ex.1007 / Page 198 of 1158
APPLE INC.

resulting 10 bytes of scan-line data packets and two bytes of vertical replication packets would encode the first 100 scan lines of the image, containing 64,000 bytes, as only 12 bytes—a considerable savings.

Figure 9-6 illustrates 1- and 2-byte vertical replication packets.

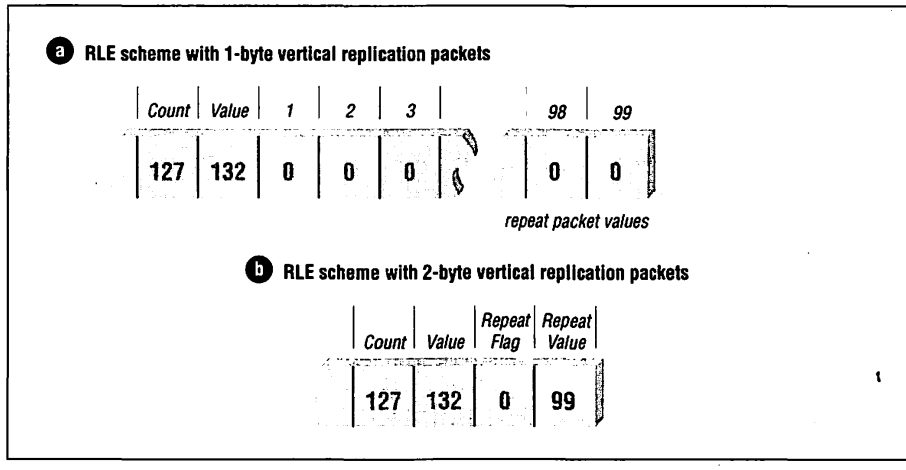


FIGURE 9-6: *RLE scheme with 1- and 2-byte vertical replication packets*

Unfortunately, definitions of vertical replication packets are application dependent. At least two common formats, WordPerfect Graphics Metafile (WPG) and GEM Raster (IMG), employ the use of repeat scan line packets to enhance data compression performance. WPG uses a simple 2-byte packet scheme, as previously described. If the first byte of an RLE packet is zero, then this is a vertical replication packet. The next byte that follows indicates the number of times to repeat the previous scan line.

The GEM Raster format is more complicated. The byte sequence, 00h 00h FFh, must appear at the beginning of an encoded scan line to indicate a vertical replication packet. The byte that follows this sequence is the number of times to repeat the previous scan line minus one.

NOTE

Many of the concepts we have covered in this section are not limited to RLE. All bitmap compression algorithms need to consider the concepts of cross-coding, sequential processing, efficient data encoding based on the data being encoded, and ways to detect and avoid negative compression.

Ex.1007 / Page 199 of 1158
APPLE INC.

For Further Information About RLE

Most books on data compression have information on run-length encoding algorithms. The following references all contain information on RLE:

Held, Gilbert, *Data Compression: Techniques and Applications, Hardware and Software Considerations*, second edition, John Wiley & Sons, New York, NY, 1987.

Lynch, Thomas D., *Data Compression Techniques and Applications*, Lifetime Learning Publications, Belmont, CA, 1985.

Nelson, Mark R., *The Data Compression Book*, M&T Books, Redwood City, CA, 1991.

Storer, James A., *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.

Lempel-Ziv-Welch (LZW) Compression

One of the most common algorithms used in computer graphics is the Lempel-Ziv-Welch, or LZW, compression scheme. This lossless method of data compression is found in several image file formats, such as GIF and TIFF, and is also part of the V.42bis modem compression standard and PostScript Level 2.

In 1977, Abraham Lempel and Jakob Ziv created the first of what we now call the LZ family of substitutional compressors. The LZ77 compression algorithms are commonly found in text compression and archiving programs, such as *compress*, *zoo*, *lha*, *pkzip*, and *arj*. The LZ78 compression algorithms are more commonly used to compress binary data, such as bitmaps.

In 1984, while working for Unisys, Terry Welch modified the LZ78 compressor for implementation in high-performance disk controllers. The result was the LZW algorithm that is commonly found today.

LZW is a general compression algorithm capable of working on almost any type of data. It is generally fast in both compressing and decompressing data and does not require the use of floating-point operations. Also, because LZW writes compressed data as bytes and not as words, LZW-encoded output can be identical on both big-endian and little-endian systems, although you may still encounter bit order and fill order problems. (See Chapter 6, *Platform Dependencies*, for a discussion of such systems.)

LZW is referred to as a *substitutional* or *dictionary-based encoding algorithm*. The algorithm builds a *data dictionary* (also called a *translation table* or *string table*) of

Ex.1007 / Page 200 of 1158
APPLE INC.

data occurring in an uncompressed data stream. Patterns of data (*substrings*) are identified in the data stream and are matched to entries in the dictionary. If the substring is not present in the dictionary, a code phrase is created based on the data content of the substring, and it is stored in the dictionary. The phrase is then written to the compressed output stream.

When a reoccurrence of a substring is identified in the data, the phrase of the substring already stored in the dictionary is written to the output. Because the phrase value has a physical size that is smaller than the substring it represents, data compression is achieved.

Decoding LZW data is the reverse of encoding. The decompressor reads a code from the encoded data stream and adds the code to the data dictionary if it is not already there. The code is then translated into the string it represents and is written to the uncompressed output stream.

LZW goes beyond most dictionary-based compressors in that it is not necessary to preserve the dictionary to decode the LZW data stream. This can save quite a bit of space when storing the LZW-encoded data. When compressing text files, LZW initializes the first 256 entries of the dictionary with the 8-bit ASCII character set (values 00h through FFh) as phrases. These phrases represent all possible single-byte values that may occur in the data stream, and all substrings are in turn built from these phrases. Because both LZW encoders and decoders begin with dictionaries initialized to these values, a decoder need not have the original dictionary and instead will build a duplicate dictionary as it decodes.

TIFF, among other file formats, applies the same method for graphic files. In TIFF, the pixel data is packed into bytes before being presented to LZW, so an LZW source byte might be a pixel value, part of a pixel value, or several pixel values, depending on the image's bit depth and number of color channels.

GIF requires each LZW input symbol to be a pixel value. Because GIF allows 1- to 8-bit deep images, there are between 2 and 256 LZW input symbols in GIF, and the LZW dictionary is initialized accordingly. It is irrelevant how the pixels might have been packed into storage originally; LZW will deal with them as a sequence of symbols.

The TIFF approach does not work very well for odd-size pixels, because packing the pixels into bytes creates byte sequences that do not match the original pixel sequences, and any patterns in the pixels are obscured. If pixel boundaries and byte boundaries agree (e.g., two 4-bit pixels per byte, or one 16-bit pixel every two bytes), then TIFF's method works well.

Ex.1007 / Page 201 of 1158
APPLE INC.

The GIF approach works better for odd-size bit depths, but it is difficult to extend it to more than eight bits per pixel because the LZW dictionary must become very large to achieve useful compression on large input alphabets.

Noise Removal and Differencing

LZW does a very good job of compressing image data with a wide variety of pixel depths. 1-, 8-, and 24-bit images all compress at least as well as they do using RLE encoding schemes. Noisy images, however, can significantly degrade the compression effectiveness of LZW. Removing noise from an image, usually by zeroing out one or two of the least significant bit planes of the image, is recommended to increase compression efficiency. In other words, if your data does not compress well in its present form, transform it to a different form that does compress well.

One method that is used to make data more "compressible" by reducing the amount of extraneous information in an image is called *differencing*. The idea is that unrelated data may be easily converted by an invertible transform into a form that can be more efficiently compressed by an encoding algorithm. Differencing accomplishes this using the fact that adjacent pixels in many continuous-tone images vary only slightly in value. If we replace the value of a pixel with the difference between the pixel and the adjacent pixel, we will reduce the amount of information stored, without losing any data.

With 1-bit monochrome and 8-bit gray-scale images, the pixel values themselves are differenced. RGB pixels must have each of their color channels differenced separately, rather than the absolute value of the RGB pixels' differences (difference red from red, green from green, and blue from blue).

Differencing is usually applied in a horizontal plane across scan lines. In the following code example, the algorithm starts at the last pixel on the first scan line of the bitmap. The difference between the last two pixels in the line is calculated, and the last pixel is set to this value. The algorithm then moves to the next to last pixel and continues up the scan line and down the bitmap until finished, as shown in the following pseudo-code:

```
/* Horizontally difference a bitmap */
for (Line = 0; Line < NumberOfLines; Line++)
    for (Pixel = NumberOfPixelsPerLine - 1; Pixel >= 1; Pixel--)
        Bitmap[Line][Pixel] -= Bitmap[Line][Pixel-1];
```

Vertical and 2D differencing may also be accomplished in the same way. The type of differencing used will have varied effectiveness depending upon the content of the image. And, regardless of the method used, differenced images compress much more efficiently using LZW.

Ex.1007 / Page 202 of 1158
APPLE INC.

Variations on the LZW Algorithm

Several variations of the LZW algorithm increase its efficiency in some applications. One common variation uses index pointers that vary in length, usually starting at 9 bits and growing upward to 12 or 13 bits. When an index pointer of a particular length has been used up, another bit is tacked on to increase precision.

Another popular variation in LZW compressors involves constantly monitoring the compression process for any drop in efficiency. If a drop is noted, the least recently used (LRU) phrases in the dictionary are discarded to make room for new phrases, or the entire dictionary is discarded and rebuilt.

The LZMW variant on the LZW compression method builds phrases by concatenating two phrases together, rather than by concatenating the current phrase and the next character of data. This causes a quicker buildup of longer strings at the cost of a more complex data dictionary.

LZW is a simple algorithm that is difficult to implement efficiently. Deciding when to discard phrases from the dictionary and even how to search the data dictionary during encoding (using a hashing or tree-based scheme) is necessary to improve efficiency and speed.

Variations on the standard LZW algorithm are more common than a programmer may realize. For example, the TIFF and GIF formats use the standard features of the LZW algorithm, such as a Clear code (the indication to discard the string table), EOF code (end of file), and a 12-bit limit on encoded symbol width. However, GIF treats each pixel value as a separate input symbol. Therefore, the size of the input alphabet, the starting compressed-symbol width, and the values of the Clear and EOF codes will vary depending on the pixel depth of the image being compressed.

GIF also stores compressed codes with the least significant bit first, regardless of the native bit order of the machine on which the algorithm is implemented. When two codes appear in the same byte, the first code is in the lower bits. When a code crosses a byte boundary, its least significant bits appear in the earlier bytes.

TIFF's LZW variation always reads 8-bit input symbols from the uncompressed data regardless of the pixel depth. Each symbol may therefore contain one pixel, more than one pixel, or only part of a pixel, depending upon the depth of the pixels in the image. TIFF always stores compressed codes with the most significant bit first, the opposite of GIF. (Don't confuse the byte-order indicator in the TIFF header, or the value of the FillOrder tag, with the bit order of the LZW compressed data. In a TIFF file, LZW-compressed data is always stored most significant bit first.)

Ex.1007 / Page 203 of 1158
APPLE INC.

TIFF LZW also contains a bit of a kludge. Compressed code widths are required to be incremented one code sooner than is really necessary. For example, the compressor changes from 9-bit to 10-bit codes after adding code 511 to its table rather than waiting until code 512 is added, thus wasting one bit.

We understand that the explanation for this practice is that the LZW implementation supplied by Aldus in Revision 5.0 of the TIFF Developer's Toolkit contained this bug, although the TIFF 5.0 specification itself specified the LZW algorithm correctly. By the time the problem was identified (by Sam Leffler, the head of the TIFF Advisory Committee), too many applications existed that used the erroneous implementation, and there was no way to identify incorrectly encoded LZW data. The solution was simply to change the TIFF specification to require this "variation" in the TIFF algorithm, rather than to change the code, break all existing TIFF LZW applications, and regard all previously created TIFF 5.0 LZW images as incorrect and useless.

LZW Legal Issues

On December 22, 1994, CompuServe Information Service announced that it had entered into a license agreement with Unisys Corporation for the use of the LZW compression/decompression method in CompuServe's GIF file format.

NOTE

This section attempts to provide information that will help you understand the legalities you may face when using the LZW compresion/decompression algorithms. We've used the most accurate information available to us as we went to press. However, we are not lawyers, nor are we employees of Unisys. Therefore, this text should not in any way be considered as a publication of Unisys Corporation, or as being approved by Unisys, or in any way obligating Unisys to enter into a license agreement based upon the terms, interpretations, and/or answers to questions provided in this text. Note that Unisys advises that Unisys licensing policies have been revised to reflect changes in the use of LZW and the needs of its existing and future licensees.

CompuServe also announced that all developers creating or modifying hardware or software technology that accessed the CompuServe Information Service and that used GIF should obtain a licensing agreement directly from

Ex.1007 / Page 204 of 1158
APPLE INC.

CompuServe and pay a royalty on each copy of their product sold. CompuServe's agreement covered only the use of GIF on CompuServe; since Unisys owned the LZW patent, Unisys claimed that any software that used GIF was subject to licensing.

The graphics and online community reacted with anger and panic. They had been misled by CompuServe to assume that the GIF file format had been freely available for unrestricted use since its invention in 1987. By 1994, GIF had risen to replace PCX as the most used file format for 8-bit color graphics. All graphics editing and display packages, including online browsers, supported the GIF format. And now, it seemed that the GIF format was not in the public domain. What was going on?

CompuServe explained that they had been approached by Unisys Corporation, which owned the patent on the LZW compression/decompression algorithm. Apparently, CompuServe had not done its homework when it included the patented LZW algorithms in the GIF file format. The result was that CompuServe had to take a license from Unisys for the use of LZW in GIF. CompuServe decided to extend a licensing requirement to everyone who was developing any GIF-using products that interfaced to CompuServe's networks. All money collected by CompuServe would essentially go to pay CompuServe's own licensing agreement with Unisys. It looked as if CompuServe was asking GIF software developers to pay for the Unisys license that CompuServe had to take because of its failure to check out whether GIF was free of patent liability.

Some people were suspicious about the timing of the CompuServe announcement. It had been known for many years in the programming community that GIF used LZW and that LZW was patented by Unisys. Yet CompuServe continued to promote the use of GIF. Unisys claimed that the company had only recently discovered that GIF used LZW. It was also a fact that the World Wide Web industry was exploding and that GIF was an integral interchange medium for transferring low-resolution graphics across the Internet.

After the smoke cleared, Unisys' LZW patent and licensing agreements held fast. Unisys softened the burden on GIF software developers by substantially reducing the license fees it had charged prior to 1995, thus offering very reasonable license fees. In addition, Unisys announced that it would not seek license fees for inadvertent infringement by GIF software products delivered prior to 1995. (However, license fees are required for updates delivered after 1995.)

People realized, upon closer examination, that it is not illegal to own, transmit, or receive GIF files (provided that no unlicensed compression and/or

Ex.1007 / Page 205 of 1158
APPLE INC.

decompression of the files occurs). It was, in fact, the implementation of the LZW algorithm that was under licensing restriction. However, because every GIF file contains LZW-encoded data, this didn't make people feel much better. Many developers swore off using GIF, while still others started grass-roots projects aimed at developing a file format to one day replace GIF. However, wide GIF usage remains and is not likely to go away any time soon, if ever.

Some History

To better understand what the Unisys LZW licensing agreement may mean to you, let's first go back a little further in history.

- In 1977, Abraham Lempel and Jakob Ziv published a paper on a universal algorithm for data compression. This was called the LZ77 compression algorithm.
- In 1978, Lempel and Ziv introduced an improved, dictionary-based compression scheme called LZ78.
- In 1981, while working for Sperry Corporation, Lempel and Ziv, along with Cohen and Eastman, filed for a patent claiming the LZ78 compression algorithm. They were granted the patent (number 4,464,650) in 1984.
- Also in 1984, while working for Sperry Corporation, Terry Welch modified the LZ78 algorithm to increase efficiency for implementation in high-performance disk controllers. The result was the LZW algorithm, which he described in an *IEEE Computer* journal article (see "For Further Information About LZW") after he left the employment of Sperry.
- In 1985, Sperry Corporation was granted a patent (number 4,558,302) for the Welch invention and implementation of the LZW data compression algorithm. Since that time, this LZW patent has been publicly available for all to see in the U.S. Patent Office and many public libraries, and is available through many online services. In addition, foreign patents with even broader scope than the U.S. patent are pending or have been granted in Canada, France, Italy, Germany, the U.K., and Japan.
- In 1986, Sperry Corporation and Burroughs merged to form Unisys. At this time, the ownership of the Sperry patent was transferred to Unisys.
- In 1987, CompuServe created the GIF file format for use in the storage and online retrieval of bitmapped graphics data. The GIF specification required the use of the LZW algorithm to compress the data stored in each GIF file. It is very possible that CompuServe did not check the patent files to determine whether the GIF format infringed on any patents, which should have

Ex.1007 / Page 206 of 1158
APPLE INC.

been done in view of their wide promotion of the GIF format. Such a check would have found the Welch LZW patent, which was then owned by Unisys. At that time, Unisys also apparently did not know that LZW was the method of compression used by the very popular GIF file format.

- In 1988, Aldus Corporation released Revision 5.0 of the TIFF file format. This revision added a new feature giving TIFF the ability to store RGB bitmapped data using either a raw format or a compressed format which used the LZW algorithm. (Although the LZW algorithm used by TIFF is considered to be "broken," it is still covered by the Unisys patent.) Since 1991, in accordance with Aldus' agreement with Unisys, Aldus has been required to place a notice regarding the Unisys patent, and its applicability to TIFF, in Aldus documentation. The 1992 release of Revision 6.0 of the TIFF specification includes this notice of the Unisys patent regarding LZW.
- In 1990, Unisys licensed Adobe for the use of the Unisys LZW patent for PostScript.
- In 1991, Unisys licensed Aldus for the use of the Unisys LZW patent in TIFF.
- In 1993, Unisys became aware that the CompuServe GIF file format uses the LZW algorithm. Negotiations began with CompuServe to create a licensing agreement for the utilization of LZW.
- In 1994, Unisys and CompuServe came to an understanding whereby the use of the LZW algorithm by Compuserve would be licensed for the application of the GIF file format in software used primarily to access the CompuServe Information Service.
- In 1995, America Online and Prodigy also entered into license agreements with Unisys for the utilization of LZW.

Since 1990 hundreds of companies have entered into LZW licensing agreements with Unisys.

For Further Information About LZW

Many books on data compression contain information on the LZ and LZW compression algorithms. The first reference below is the definitive source for a very general explanation about the LZW algorithm itself and does not focus specifically on bitmap image data:

Welch, T. A., "A Technique for High Performance Data Compression," *IEEE Computer*, vol. 17, no. 6, June 1984.

Ex.1007 / Page 207 of 1158
APPLE INC.

The TIFF specification (both revisions 5.0 and 6.0) contains an explanation of the TIFF variation on LZW compression. Refer to the "For Further Information" section of the TIFF article in Part Two of this book for information and see the CD-ROM for the specification itself.

The following articles and manuscripts are also specifically related to LZW:

Bell, Timothy C., "Better OPM/L Text Compression," *IEEE Transactions on Communications*, vol. 34, no. 12, December 1986, 1176–1182.

Bernstein, Daniel J., *Y coding, Draft 4b*, March 19, 1991, manuscript part of the Yabba Y Coding package.

Blackstock, Steve, *LZW and GIF Explained*, manuscript in the public domain, 1987.

Montgomery, Bob, *LZW Compression Used to Encode/Decode a GIF File*, manuscript in the public domain, 1988.

Nelson, Mark R., "LZW Data Compression," *Dr. Dobbs Journal*, October 1989, pp. 29–36.

Phillips, Dwayne, "LZW Data Compression," *The Computer Applications Journal*, Circuit Cellar Ink, vol. 27, June/July 1992, pp. 36–48.

Rodriguez, Karen, "Graphics file format patent Unisys seeks royalties from GIF developers," *InfoWorld*, vol. 17, January 9, 1995, p. 3.

Thomborson, Clark, "The V.42bis standard for data-compressing modems," *IEEE Micro*, October 1992, pp. 41–53.

Ziv, J., and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, 1977, pp. 337–343.

Ziv, J., and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, September 1978.

You can get additional information about LZW, and find out about licensing of LZW, by contacting Unisys:

Welch Patent Licensing Department
Unisys Corporation
Mail Stop C1SW19
P.O. Box 500
Blue Bell, PA 19424 USA
FAX: 215-986-3090
Email: *lzw_info@unisys.com*

Ex.1007 / Page 208 of 1158
APPLE INC.

General licensing information may also be obtained from the homepage of the Unisys Web server:

http://www.unisys.com

In particular, see:

http://www.unisys.com/LeadStory/lzwterms.html
http://www.unisys.com/LeadStory/lzwfaq.html

The *comp.graphics.misc* and *comp.compression* USENET newsgroups contain frequent discussions of LZW technical issues and some discussions of the patent issues. The official newsgroup where much discussion takes place is *alt.gif-agreement.* You might also find information on the *misc.legal.computing, misc.int-property,* and *comp.patents* newsgroups.

Be sure to check out the compression and graphics file formats FAQs as well. Both contain a substantial amount of information about LZW and the patent issues.

You can get a copy of the actual LZW patent from the U.S. Patent Office. The patent is also available at many Internet sites, including:

ftp://cs.columbia.edu/archives/mirror2/world-info/obi/USPatents/lzw-patent.gz
ftp://ftp.std.com/obi/USPatents/lzw-patent.Z
ftp://ftp.uu.net/doc/literary/obi/USPatents/lzw-patent.Z
ftp://gatekeeper.dec.com/.8/misc/lzw-patent.Z

CCITT (Huffman) Encoding

Many facsimile and document imaging file formats support a form of lossless data compression often described as CCITT encoding. The CCITT (International Telegraph and Telephone Consultative Committee) is a standards organization that has developed a series of communications protocols for the facsimile transmission of black-and-white images over telephone lines and data networks. These protocols are known officially as the CCITT T.4 and T.6 standards but are more commonly referred to as CCITT Group 3 and Group 4 compression, respectively.

Sometimes CCITT encoding is referred to, not entirely accurately, as Huffman encoding. Huffman encoding is a simple compression algorithm introduced by David Huffman in 1952. CCITT 1-dimensional encoding, described in a subsection below, is a specific type of Huffman encoding. The other types of CCITT encodings are not, however, implementations of the Huffman scheme.

Ex.1007 / Page 209 of 1158
APPLE INC.

Group 3 and Group 4 encodings are compression algorithms that are specifically designed for encoding 1-bit image data. Many document and FAX file formats support Group 3 compression, and several, including TIFF, also support Group 4.

Group 3 encoding was designed specifically for bi-level, black-and-white image data telecommunications. All modern FAX machines and FAX modems support Group 3 facsimile transmissions. Group 3 encoding and decoding is fast, maintains a good compression ratio for a wide variety of document data, and contains information that aids a Group 3 decoder in detecting and correcting errors without special hardware.

Group 4 is a more efficient form of bi-level compression that has almost entirely replaced the use of Group 3 in many conventional document image storage systems. (An exception is facsimile document storage systems where original Group 3 images are required to be stored in an unaltered state.)

Group 4 encoded data is approximately half the size of 1-dimensional Group 3–encoded data. Although Group 4 is fairly difficult to implement efficiently, it encodes at least as fast as Group 3 and in some implementations decodes even faster. Also, Group 4 was designed for use on data networks, so it does not contain the synchronization codes used for error detection that Group 3 does, making it a poor choice for an image transfer protocol.

Group 4 is sometimes confused with the IBM MMR (Modified Modified READ) compression method. In fact, Group 4 and MMR are almost exactly the same algorithm and achieve almost identical compression results. IBM released MMR in 1979 with the introduction of its Scanmaster product before Group 4 was standardized. MMR became IBM's own document compression standard and is still used in many IBM imaging systems today.

Document-imaging systems that store large amounts of facsimile data have adopted these CCITT compression schemes to conserve disk space. CCITT-encoded data can be decompressed quickly for printing or viewing (assuming that enough memory and CPU resources are available). The same data can also be transmitted using modem or facsimile protocol technology without needing to be encoded first.

The CCITT algorithms are non-adaptive. That is, they do not adjust the encoding algorithm to encode each bitmap with optimal efficiency. They use a fixed table of code values that were selected according to a reference set of documents containing both text and graphics. The reference set of documents were considered to be representative of documents that would be transmitted by facsimile.

Ex.1007 / Page 210 of 1158
APPLE INC.

Group 3 normally achieves a compression ratio of 5:1 to 8:1 on a standard 200-dpi (204×196 dpi), A4-sized document. Group 4 results are roughly twice as efficient as Group 3, achieving compression ratios upwards of 15:1 with the same document. Claims that the CCITT algorithms are capable of far better compression on standard business documents are exaggerated—largely by hardware vendors.

Because the CCITT algorithms have been optimized for type and handwritten documents, it stands to reason that images radically different in composition will not compress very well. This is all too true. Bi-level bitmaps that contain a high frequency of short runs, as typically found in digitally halftoned continuous-tone images, do not compress as well using the CCITT algorithms. Such images will usually result in a compression ratio of 3:1 or even lower, and many will actually compress to a size larger than the original.

The CCITT actually defines three algorithms for the encoding of bi-level image data:

- Group 3 One-Dimensional (G31D)
- Group 3 Two-Dimensional (G32D)
- Group 4 Two-Dimensional (G42D)

G31D is the simplest of the algorithms and the easiest to implement. For this reason, it is discussed in its entirety in the first subsection below. G32D and G42D are much more complex in their design and operation and are described only in general terms below.

The Group 3 and Group 4 algorithms are standards and therefore produce the same compression results for everybody. If you have heard any claims made to the contrary, it is for one of these reasons:

- Non-CCITT test images are being used as benchmarks.
- Proprietary modifications have been made to the algorithm.
- Pre- or post-processing is being applied to the encoded image data.
- You have been listening to a misinformed salesperson.

Group 3 One-Dimensional (G31D)

Group 3 One-Dimensional encoding (G31D) is a variation of the Huffman keyed compression scheme. A bi-level image is composed of a series of black-and-white 1-bit pixel runs of various lengths (1 = black and 0 = white). A Group 3 encoder determines the length of a pixel run in a scan line and outputs a

Ex.1007 / Page 211 of 1158
APPLE INC.

variable-length binary code word representing the length and color of the run. Because the code word output is shorter than the input, pixel data compression is achieved.

The run-length code words are taken from a predefined table of values representing runs of black or white pixels. This table is part of the T.4 specification and is used to encode and decode all Group 3 data.

The size of the code words were originally determined by the CCITT, based statistically on the average frequency of black-and-white runs occurring in typical type and handwritten documents. The documents included line art and were written in several different languages. Run lengths that occur more frequently are assigned smaller code words while run lengths that occur less frequently are assigned larger code words.

In printed and handwritten documents, short runs occur more frequently than long runs. Two- to 4-pixel black runs are the most frequent in occurrence. The maximum size of a run length is bounded by the maximum width of a Group 3 scan line.

Run lengths are represented by two types of code words: *makeup* and *terminating*. An encoded pixel run is made up of zero or more makeup code words and a terminating code word. Terminating code words represent shorter runs, and makeup codes represent longer runs. There are separate terminating and makeup code words for both black and white runs.

Pixel runs with a length of 0 to 63 are encoded using a single terminating code. Runs of 64 to 2623 pixels are encoded by a single makeup code and a terminating code. Run lengths greater than 2623 pixels are encoded using one or more makeup codes and a terminating code. The run length is the sum of the length values represented by each code word.

Here are some examples of several different encoded runs:

- A run of 20 black pixels would be represented by the terminating code for a black run length of 20. This reduces a 20-bit run to the size of an 11-bit code word, a compression ratio of nearly 2:1. This is illustrated in Figure 9-7, a.
- A white run of 100 pixels would be encoded using the makeup code for a white run length of 64 pixels followed by the terminating code for a white run length of 36 pixels (64 + 36 = 100). This encoding reduces 100 bits to 13 bits, or a compression ratio of over 7:1. This is illustrated in Figure 9-7, b.

Ex.1007 / Page 212 of 1158
APPLE INC.

- A run of 8800 black pixels would be encoded as three makeup codes of 2560 black pixels (7680 pixels), a makeup code of 1088 black pixels, followed by the terminating code for 32 black pixels (2560 + 2560 + 2560 + 1088 + 32 = 8800). In this case, we will have encoded 8800 run-length bits into five code words with a total length of 61 bits, for an approximate compression ratio of 144:1. This is illustrated in Figure 9-7, c.

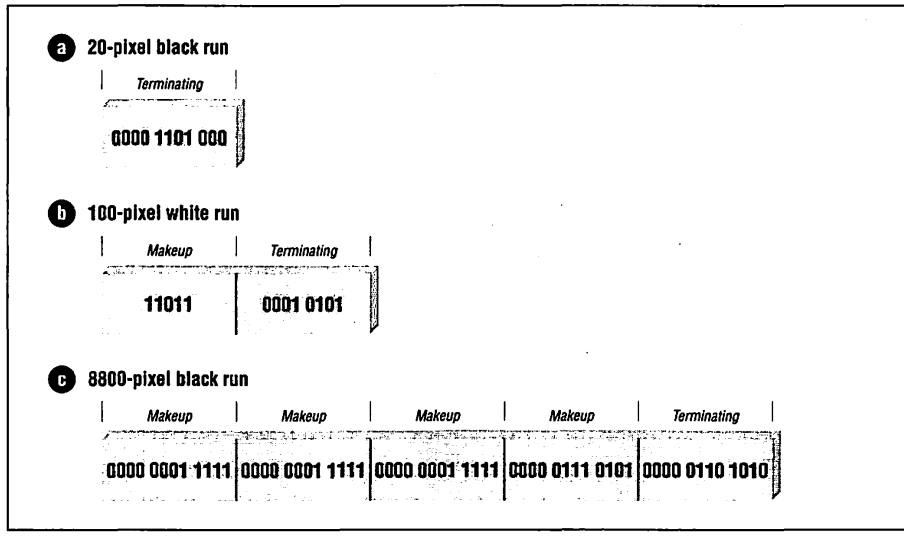


FIGURE 9-7: *CCITT Group 3 encoding*

The use of run lengths encoded with multiple makeup codes has become a de facto extension to Group 3, because such encoders are necessary for images with higher resolutions. And while most Group 3 decoders do support this extension, do not expect them to do so in all cases.

Decoding Group 3 data requires methods different from most other compression schemes. Because each code word varies in length, the encoded data stream must be read one bit at a time until a code word is recognized. This can be a slow and tedious process at best. To make this job easier, a state table can be used to process the encoded data one byte at a time. This is the quickest and most efficient way to implement a CCITT decoder.

All scan lines are encoded to begin with a white run-length code word (most document image scan lines begin with white run lengths). If an actual scan line begins with a black run, a zero-length white run-length code word will be prepended to the scan line.

Ex.1007 / Page 213 of 1158
APPLE INC.

A decoder keeps track of the color of the run it is decoding. Comparing the current bit pattern to values in the opposite color bit table is wasteful. That is, if a black run is being decoded, there is no reason to check the table for white run-length codes.

Several special code words are also defined in a Group 3–encoded data stream. These codes are used to provide synchronization in the event that a phone transmission experiences a burst of noise. By recognizing this special code, a CCITT decoder may identify transmission errors and attempt to apply a recovery algorithm that approximates the lost data.

The EOL code is a 12-bit code word that begins each line in a Group 3 transmission. This unique code word is used to detect the start/end of a scan line during the image transmission. If a burst of noise temporarily corrupts the signal, a Group 3 decoder throws away the unrecognized data it receives until it encounteres an EOL code. The decoder would then start receiving the transmission as normal again, assuming that the data following the EOL is the beginning of the next scan line. The decoder might also replace the bad line with a predefined set of data, such as a white scan line.

A decoder also uses EOL codes for several purposes. It uses them to keep track of the width of a decoded scan line. (An incorrect scan-line width may be an error, or it may be an indication to pad with white pixels to the EOL.) In addition, it uses EOL codes to keep track of the number of scan lines in an image, in order to detect a short image. If it finds one, it pads the remaining length with scan lines of all white pixels. A Group 3 EOL code is illustrated in Figure 9-8.

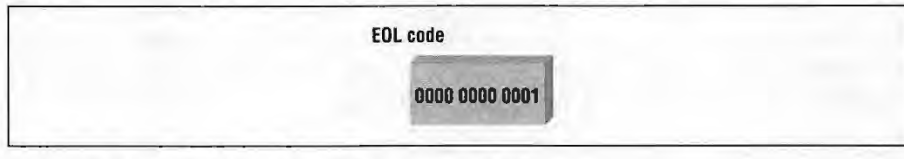


FIGURE 9-8: *CCITT Group 3 encoding (EOL code)*

Most FAX machines transmit data of an "unlimited length," in which case the decoder cannot detect how long the image is supposed to be. Also, it is faster not to transmit the all-white space at the end of a page, and many FAX machines stop when they detect that the rest of a page is all white; they expect the receiver to do white padding to the end of the negotiated page size.

When Group 3 data is encapsulated in an image file, information regarding the length and width of the image is typically stored in the image file header and is read by the decoder prior to decoding.

Ex.1007 / Page 214 of 1158
APPLE INC.

Group 3 message transmissions are terminated by a return to control (RTC) code that is appended to the end of every Group 3 data stream and is used to indicate the end of the message transmission. An RTC code word is simply six EOL codes occurring consecutively. The RTC is actually part of the facsimile protocol and not part of the encoded message data. It is used to signal the receiver that it should drop the high-speed message carrier and listen on the low-speed carrier for the post-page command. A Group 3 RTC code is illustrated in Figure 9-9.

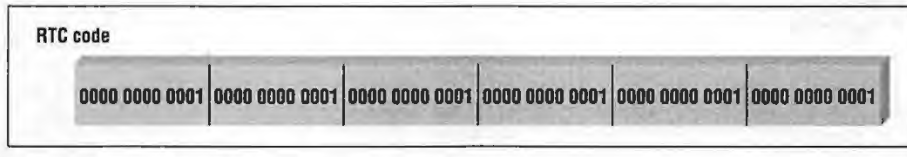


FIGURE 9-9: *CCITT Group 3 encoding (RTC code)*

A fill (FILL) is not actually a code word but a run of one or more zero bits that occurs between the encoded scan-line data and the EOL code (but never in the encoded scan line itself). Fill bits are used to pad out the length of an encoded scan line to increase the transmission time of the line to a required length. Fill bits may also be used to pad the RTC code word out to end on a byte boundary.

TIFF Compression Type 2

The TIFF compression Type 2 scheme (in which the compression tag value is equal to 2) is a variation of CCITT G31D encoding. The TIFF Type 3 and TIFF Type 4 compression methods follow exactly the CCITT Group 3 and Group 4 specifications, respectively. Type 2 compression, on the other hand, implements Group 3 encoding but does not use EOL or RTC code words. For this reason, TIFF Type 2 compression is also called "Group 3, No EOLs." Also, fill bits are never used except to pad out the last byte in a scan line to the next byte boundary.

These modifications were incorporated into the TIFF specification because EOL and RTC codes are not needed to read data stored on tape or disk. A typical letter-size image (1728×2200 pixels) would contain 26,484 bits (3310.5 bytes) of EOL and RTC information. When storing Group 3 data to a file, the following are not needed:

- The initial 12-bit EOL
- The 12 EOL bits per scan line

Ex.1007 / Page 215 of 1158
APPLE INC.

- The 72 RTC bits tacked onto the end of each image

Conventional Group 3 decoders cannot handle these modifications and will either refuse to read the TIFF Type 2–encoded data or will simply return a stream of decoded garbage. However, many decoders have been designed to accept these Group 3 trivial "modifications" and have no problems reading this type of data. Group 3 encoding is illustrated in Figure 9-10.

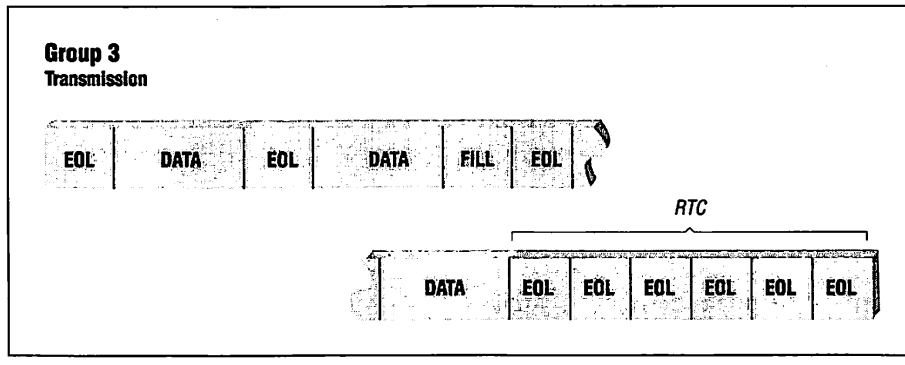


FIGURE 9-10: *Group 3 CCITT encoding (TIFF Compression Type 2)*

TIFF Class F

There is nearly one facsimile file format for every brand of computer FAX hardware and software made. Many compress the facsimile data using RLE (presumably so it will be quicker to display) or store it in its original Group 3 encoding. Perhaps the most widely used FAX file format is the unofficial TIFF Class F format. (See the TIFF article in Part Two of this book for more information about TIFF Class F.)

Even with the latest release of TIFF, revision 6.0, Class F has never been officially included in the standard, despite the wishes of the TIFF Advisory Council. The reason for this is that Aldus feels that supporting applications that require facsimile data storage and retrieval is outside of the scope of TIFF. (TIFF was designed primarily with scanners and desktop publishing in mind.) This is too bad, considering that one of TIFF's main goals is to aid in promoting image data interchangeability between hardware platforms and software applications.

Group 3 Two-Dimensional (G32D)

Group 3 One-Dimensional (G31D) encoding, which we've discussed above, encodes each scan line independent of the other scan lines. Only one run

Ex.1007 / Page 216 of 1158
APPLE INC.

length at a time is considered during the encoding and decoding process. The data occurring before and after each run length is not important to the encoding step; only the data occurring in the present run is needed.

With Group 3 Two-Dimensional (G32D) encoding, on the other hand, the way a scan line is encoded may depend on the immediately preceding scan-line data. Many images have a high degree of vertical coherence (redundancy). By describing the differences between two scan lines, rather than describing the scan line contents, 2D encoding achieves better compression.

The first pixel of each run length is called a *changing element.* Each changing element marks a color transition within a scan line (the point where a run of one color ends and a run of the next color begins).

The position of each changing element in a scan line is described as being a certain number of pixels from a changing element in the current, coding line (horizontal coding is performed) or in the preceding, reference line (vertical coding is performed). The output codes used to describe the actual positional information are called Relative Element Address Designate (READ) codes.

Shorter code words are used to describe the color transitions that are less than four pixels away from each other on the code line or the reference line. Longer code words are used to describe color transitions lying a greater distance from the current changing element.

2D encoding is more efficient than 1-dimensional because the usual data that is compressed (typed or handwritten documents) contains a high amount of 2D coherence.

Because a G32D-encoded scan line is dependent on the correctness of the preceding scan line, an error, such as a burst of line noise, can affect multiple, 2-dimensionally encoded scan lines. If a transmission error corrupts a segment of encoded scan line data, that line cannot be decoded. But, worse still, all scan lines occurring after it also decode improperly.

To minimize the damage created by noise, G32D uses a variable called a K factor and 2-dimensionally encodes K-1 lines following a 1-dimensionally encoded line. If corruption of the data transmission occurs, only K-1 scan lines of data will be lost. The decoder will be able to resync the decoding at the next available EOL code.

The typical value for K is 2 or 4. G32D data that is encoded with a K value of 4 appears as a single block of data. Each block contains three lines of 2D scan-line data followed by a scan line of 1-dimensionally encoded data.

Ex.1007 / Page 217 of 1158
APPLE INC.

The K variable is not normally used in decoding the G32D data. Instead, the EOL code is modified to indicate the algorithm used to encode the line following it. If a 1 bit is appended to the EOL code, the line following is 1-dimensionally encoded; if a 0 bit is appended, the line following the EOL code is 2-dimensionally encoded. All other transmission code word markers (FILL and RTC) follow the same rule as in G31D encoding. K is only needed in decoding if regeneration of the previous 1-dimensionally encoded scan line is necessary for error recovery.

Group 4 Two-Dimensional (G42D)

Group 4 Two-Dimensional (G42D) encoding was developed from the G32D algorithm as a better 2D compression scheme—so much better, in fact, that Group 4 has almost completely replaced G32D in commercial use.

Group 4 encoding is identical to G32D encoding except for a few modifications. Group 4 is basically the G32D algorithm with no EOL codes and a K variable set to infinity. Group 4 was designed specifically to encode data residing on disk drives and data networks. The built-in transmission error detection/correction found in Group 3 is therefore not needed by Group 4 data.

The first reference line in Group 4 encoding is an imaginary scan line containing all white pixels. In G32D encoding, the first reference line is the first scan line of the image. In Group 4 encoding, the RTC code word is replaced by an end of facsimile block (EOFB) code, which consists of two consecutive Group 3 EOL code words. Like the Group 3 RTC, the EOFB is also part of the transmission protocol and not actually part of the image data. Also, Group 4–encoded image data may be padded out with fill bits after the EOFB to end on a byte boundary.

Group 4 encoding will usually result in an image compressed twice as small as if it were done with G31D encoding. The main tradeoff is that Group 4 encoding is more complex and requires more time to perform. When implemented in hardware, however, the difference in execution speed between the Group 3 and Group 4 algorithms is not significant, which usually makes Group 4 a better choice in most imaging system implementations.

Tips for Designing CCITT Encoders and Decoders

Here are some general guidelines to follow if you are using the CCITT encoding method to encode or decode.

- Ignore bits occurring after the RTC or EOFB markers. These markers indicate the end of the image data, and all bits occurring after them can be considered filler.

Ex.1007 / Page 218 of 1158
APPLE INC.

- You must know the number of pixels in a scan line before decoding. Any row that decodes to fewer or greater pixels than expected is normally considered corrupt, and further decoding of the image block (2D encoding only) should not be attempted. Some encoding schemes will produce short scan lines if the line contains all white pixels. The decoder is expected to realize this and pad out the entire line as a single white run.
- Be aware that decoded scan-line widths will not always be a multiple of eight, and decoders should not expect byte-boundary padding to always occur. Robust decoders should be able to read non-byte-aligned data.
- If a decoder encounters an RTC or EOFB marker before the expected number of scan lines has been decoded, assume that the remaining scan lines are all white. If the expected number of scan lines has been decoded, but an RTC or EOFB has not been encountered, stop decoding. A decoder should then produce a warning that an unusual condition has been detected.
- Note that a well-designed CCITT decoder should be able to handle the typical color-sex and bit-sex problems associated with 1-bit data, as described below:

  The CCITT defines a pixel value of 0 as white and a pixel value of 1 as black. Many bitmaps, however, may be stored using the opposite pixel color values and the decoder would interpret a "negative" of the image in this case (a color-sex problem). Different machine architectures also store the bits within a byte in different ways. Some store the most significant bit first, and some store the least significant bit first. If bitmap data is read in the opposite format from the one in which it was stored, the image will appear fragmented and disrupted (a bit-sex problem). To prevent these problems, always design a CCITT decoder that is capable of reading data using either of the color-sex and bit-sex schemes to suit the requirements of the user.

For Further Information About CCITT

The original specifications for the CCITT Group 3 and Group 4 algorithms are in CCITT (1985) Volume VII, Fascicle VII.3, Recommendations T.4 and T.6:

"Standardization of Group 3 Facsimile Apparatus for Document Transmission," *Recommendation T.4, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services*, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, Switzerland, 1985, pp. 16–31.

Ex.1007 / Page 219 of 1158
APPLE INC.

"Facsimile Coding Schemes and Coding Control Functions for Group 4 Facsimile Apparatus," *Recommendation T.6, Volume VII, Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services*, The International Telegraph and Telephone Consultative Committee (CCITT), Geneva, Switzerland, 1985, pp. 40–48.

The latest specification is CCITT (1992) Volume VII, Fascicle VII.3, Recommendations T.0 through T.63.

Both the CCITT and ANSI documents may be obtained from the following source:

American National Standards Institute, Inc.
Attn: Sales
1430 Broadway
New York, NY 10018 USA
Voice: 212-642-4900

See also the following references. (For information on getting RFCs [Requests for Comments], send email to *rfc-info@isi.edu* with a subject line of "getting rfcs" and a body of "help: ways _to_get_rfcs".)

Hunter, R. and A.H. Robinson, "International Digital Facsimile Coding Standards," *Proceedings of the IEEE*, vol. 68, no. 7, July 1980, pp. 854–867.

RFC 804—CCITT Draft Recommendation T.4 (Standardization of Group 3 Facsimile Apparatus for Document Transmission).

RFC 1314—A File Format for the Exchange of Images in the Internet.

FAX: Digital Facsimile Technology & Applications, McConnell, Artech House, Norwood, MA; second edition, 1992.

Marking, Michael P., "Decoding Group 3 Images," *The C Users Journal*, June 1990, pp. 45–54.

Information on MMR encoding may be found in the following references:

"Binary-image-manipulation Algorithms in the Image View Facility," *IBM Journal of Research and Development*, vol. 31, no. 1, January 1987.

Information on Huffman encoding may be found in the following references:

Huffman, David, "A Method for the Construction of Minimum Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, 1952, pp. 1098–1101.

Kruger, Anton, "Huffman Data Compression," *C Gazette*, vol. 5, no. 4, 1991, pp.71–77.

Ex.1007 / Page 220 of 1158
APPLE INC.

JPEG Compression

One of the hottest topics in image compression technology today is JPEG. The acronym JPEG stands for the Joint Photographic Experts Group, a standards committee that had its origins within the International Standard Organization (ISO). In 1982, the ISO formed the Photographic Experts Group (PEG) to research methods of transmitting video, still images, and text over ISDN (Integrated Services Digital Network) lines. PEG's goal was to produce a set of industry standards for the transmission of graphics and image data over digital communications networks.

In 1986, a subgroup of the CCITT began to research methods of compressing color and gray-scale data for facsimile transmission. The compression methods needed for color facsimile systems were very similar to those being researched by PEG. It was therefore agreed that the two groups should combine their resources and work together toward a single standard.

In 1987, the ISO and CCITT combined their two groups into a joint committee that would research and produce a single standard of image data compression for both organizations to use. This new committee was JPEG.

Although the creators of JPEG might have envisioned a multitude of commercial applications for JPEG technology, a consumer public made hungry by the marketing promises of imaging and multimedia technology are benefiting greatly as well. Most previously developed compression methods do a relatively poor job of compressing continuous-tone image data; that is, images containing hundreds or thousands of colors taken from real-world subjects. And very few file formats can support 24-bit raster images.

GIF, for example, can store only images with a maximum pixel depth of eight bits, for a maximum of 256 colors. And its LZW compression algorithm does not work very well on typical scanned image data. The low-level noise commonly found in such data defeats LZW's ability to recognize repeated patterns.

Both TIFF and BMP are capable of storing 24-bit data, but in their pre-JPEG versions are capable of using only encoding schemes (LZW and RLE, respectively) that do not compress this type of image data very well.

JPEG provides a compression method that is capable of compressing continuous-tone image data with a pixel depth of 6 to 24 bits with reasonable speed and efficiency. And although JPEG itself does not define a standard image file format, several have been invented or modified to fill the needs of JPEG data storage.

Ex.1007 / Page 221 of 1158
APPLE INC.

JPEG in Perspective

Unlike all of the other compression methods described so far in this chapter, JPEG is not a single algorithm. Instead, it may be thought of as a toolkit of image compression methods that may be altered to fit the needs of the user. JPEG may be adjusted to produce very small, compressed images that are of relatively poor quality in appearance but still suitable for many applications. Conversely, JPEG is capable of producing very high-quality compressed images that are still far smaller than the original uncompressed data.

JPEG is also different in that it is primarily a lossy method of compression. Most popular image format compression schemes, such as RLE, LZW, or the CCITT standards, are lossless compression methods. That is, they do not discard any data during the encoding process. An image compressed using a lossless method is guaranteed to be identical to the original image when uncompressed.

Lossy schemes, on the other hand, throw useless data away during encoding. This is, in fact, how lossy schemes manage to obtain superior compression ratios over most lossless schemes. JPEG was designed specifically to discard information that the human eye cannot easily see. Slight changes in color are not perceived well by the human eye, while slight changes in intensity (light and dark) are. Therefore JPEG's lossy encoding tends to be more frugal with the gray-scale part of an image and to be more frivolous with the color.

JPEG was designed to compress color or gray-scale continuous-tone images of real-world subjects: photographs, video stills, or any complex graphics that resemble natural subjects. Animations, ray tracing, line art, black-and-white documents, and typical vector graphics don't compress very well under JPEG and shouldn't be expected to. And, although JPEG is now used to provide motion video compression, the standard makes no special provision for such an application.

The fact that JPEG is lossy and works only on a select type of image data might make you ask, "Why bother to use it?" It depends upon your needs. JPEG is an excellent way to store 24-bit photographic images, such as those used in imaging and multimedia applications. JPEG 24-bit (16 million color) images are superior in appearance to 8-bit (256 color) images on a VGA display and are at their most spectacular when using 24-bit display hardware (which is now quite inexpensive).

The amount of compression achieved depends upon the content of the image data. A typical photographic-quality image may be compressed from 20:1 to

Ex.1007 / Page 222 of 1158
APPLE INC.

25:1 without experiencing any noticeable degradation in quality. Higher compression ratios will result in image files that differ noticeably from the original image but still have an overall good image quality. And achieving a 20:1 or better compression ratio in many cases not only saves disk space, but also reduces transmission time across data networks and phone lines.

An end user can "tune" the quality of a JPEG encoder using a parameter sometimes called a *quality setting* or a *Q factor*. Although different implementations have varying scales of Q factors, a range of 1 to 100 is typical. A factor of 1 produces the smallest, worst quality images; a factor of 100 produces the largest, best quality images. The optimal Q factor depends on the image content and is therefore different for every image. The art of JPEG compression is finding the lowest Q factor that produces an image that is visibly acceptable, and preferably as close to the original as possible.

The JPEG library supplied by the Independent JPEG Group (included on the CD-ROM that accompanies this book) uses a quality setting scale of 1 to 100. To find the optimal compression for an image using the JPEG library, follow these steps:

1. Encode the image using a quality setting of 75 (-Q 75).
2. If you observe unacceptable defects in the image, increase the value, and re-encode the image.
3. If the image quality is acceptable, decrease the setting until the image quality is barely acceptable. This will be the optimal quality setting for this image.
4. Repeat this process for every image you have (or just encode them all using a quality setting of 75).

JPEG isn't always an ideal compression solution. There are several reasons:

- As we have said, JPEG doesn't fit every compression need. Images containing large areas of a single color do not compress very well. In fact, JPEG will introduce "artifacts" into such images that are visible against a flat background, making them considerably worse in appearance than if you used a conventional lossless compression method. Images of a "busier" composition contain even worse artifacts, but they are considerably less noticeable against the image's more complex background.
- JPEG can be rather slow when it is implemented only in software. If fast decompression is required, a hardware-based JPEG solution is your best bet, unless you are willing to wait for a faster software-only solution to come along or buy a faster computer.

Ex.1007 / Page 223 of 1158
APPLE INC.

- JPEG is not trivial to implement. It is not likely you will be able to sit down and write your own JPEG encoder/decoder in a few evenings. We recommend that you obtain a third-party JPEG library, rather than writing your own.
- JPEG is not supported by very many file formats. The formats that do support JPEG are all fairly new and can be expected to be revised at frequent intervals.

Baseline JPEG

The JPEG specification defines a minimal subset of the standard called baseline JPEG, which all JPEG-aware applications are required to support. This baseline uses an encoding scheme based on the Discrete Cosine Transform (DCT) to achieve compression. DCT is a generic name for a class of operations identified and published some years ago. DCT-based algorithms have since made their way into various compression methods.

DCT-based encoding algorithms are always lossy by nature. DCT algorithms are capable of achieving a high degree of compression with only minimal loss of data. This scheme is effective only for compressing continuous-tone images in which the differences between adjacent pixels are usually small. In practice, JPEG works well only on images with depths of at least four or five bits per color channel. The baseline standard actually specifies eight bits per input sample. Data of lesser bit depth can be handled by scaling it up to eight bits per sample, but the results will be bad for low-bit-depth source data, because of the large jumps between adjacent pixel values. For similar reasons, colormapped source data does not work very well, especially if the image has been dithered.

The JPEG compression scheme is divided into the following stages:

1. Transform the image into an optimal color space.
2. Downsample chrominance components by averaging groups of pixels together.
3. Apply a Discrete Cosine Transform (DCT) to blocks of pixels, thus removing redundant image data.
4. Quantize each block of DCT coefficients using weighting functions optimized for the human eye.
5. Encode the resulting coefficients (image data) using a Huffman variable word-length algorithm to remove redundancies in the coefficients.

Figure 9-11 summarizes these steps, and the following subsections look at each of them in turn. Note that JPEG decoding performs the reverse of these steps.

Ex.1007 / Page 224 of 1158
APPLE INC.

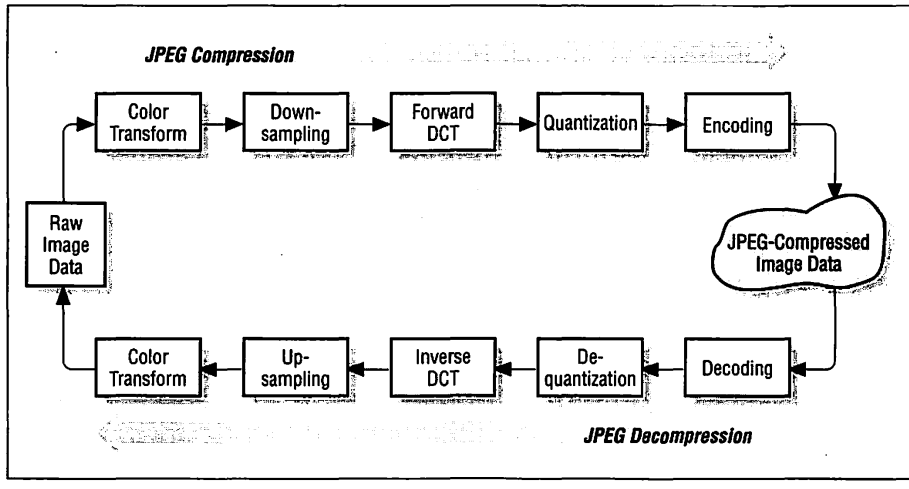


FIGURE 9-11: *JPEG compression and decompression*

Transform the image

The JPEG algorithm is capable of encoding images that use any type of color space. JPEG itself encodes each component in a color model separately, and it is completely independent of any color-space model, such as RGB, HSI, or CMY. The best compression ratios result if a luminance/chrominance color space, such as YUV or YCbCr, is used. (See Chapter 2 for a description of these color spaces.)

Most of the visual information to which human eyes are most sensitive is found in the high-frequency, gray-scale, luminance component (Y) of the YCbCr color space. The other two chrominance components (Cb and Cr) contain high-frequency color information to which the human eye is less sensitive. Most of this information can therefore be discarded.

In comparison, the RGB, HSI, and CMY color models spread their useful visual image information evenly across each of their three color components, making the selective discarding of information very difficult. All three color components would need to be encoded at the highest quality, resulting in a poorer compression ratio. Gray-scale images do not have a color space as such and therefore do not require transforming.

Ex.1007 / Page 225 of 1158
APPLE INC.

Downsample chrominance components

The simplest way of exploiting the eye's lesser sensitivity to chrominance information is simply to use fewer pixels for the chrominance channels. For example, in an image nominally 1000×1000 pixels, we might use a full 1000×1000 luminance pixels but only 500×500 pixels for each chrominance component. In this representation, each chrominance pixel covers the same area as a 2×2 block of luminance pixels. We store a total of six pixel values for each 2×2 block (four luminance values, one each for the two chrominance channels), rather than the twelve values needed if each component is represented at full resolution. Remarkably, this 50 percent reduction in data volume has almost no effect on the perceived quality of most images. Equivalent savings are not possible with conventional color models such as RGB, because in RGB each color channel carries some luminance information and so any loss of resolution is quite visible.

When the uncompressed data is supplied in a conventional format (equal resolution for all channels), a JPEG compressor must reduce the resolution of the chrominance channels by *downsampling*, or averaging together groups of pixels. The JPEG standard allows several different choices for the sampling ratios, or relative sizes, of the downsampled channels. The luminance channel is always left at full resolution (1:1 sampling). Typically both chrominance channels are downsampled 2:1 horizontally and either 1:1 or 2:1 vertically, meaning that a chrominance pixel covers the same area as either a 2×1 or a 2×2 block of luminance pixels. JPEG refers to these downsampling processes as 2h1v and 2h2v sampling, respectively.

Another notation commonly used is 4:2:2 sampling for 2h1v and 4:2:0 sampling for 2h2v; this notation derives from television customs (color transformation and downsampling have been in use since the beginning of color TV transmission). 2h1v sampling is fairly common because it corresponds to National Television Standards Committee (NTSC) standard TV practice, but it offers less compression than 2h2v sampling, with hardly any gain in perceived quality.

Apply a Discrete Cosine Transform

The image data is divided up into 8×8 blocks of pixels. (From this point on, each color component is processed independently, so a "pixel" means a single value, even in a color image.) A DCT is applied to each 8×8 block. DCT converts the spatial image representation into a frequency map: the low-order or "DC" term represents the average value in the block, while successive higher-order ("AC") terms represent the strength of more and more rapid changes

Ex.1007 / Page 226 of 1158
APPLE INC.

across the width or height of the block. The highest AC term represents the strength of a cosine wave alternating from maximum to minimum at adjacent pixels.

The DCT calculation is fairly complex; in fact, this is the most costly step in JPEG compression. The point of doing it is that we have now separated out the high- and low-frequency information present in the image. We can discard high-frequency data easily without losing low-frequency information. The DCT step itself is lossless except for roundoff errors.

Quantize each block

To discard an appropriate amount of information, the compressor divides each DCT output value by a "quantization coefficient" and rounds the result to an integer. The larger the quantization coefficient, the more data is lost, because the actual DCT value is represented less and less accurately. Each of the 64 positions of the DCT output block has its own quantization coefficient, with the higher-order terms being quantized more heavily than the low-order terms (that is, the higher-order terms have larger quantization coefficients). Furthermore, separate quantization tables are employed for luminance and chrominance data, with the chrominance data being quantized more heavily than the luminance data. This allows JPEG to exploit further the eye's differing sensitivity to luminance and chrominance.

It is this step that is controlled by the "quality" setting of most JPEG compressors. The compressor starts from a built-in table that is appropriate for a medium-quality setting and increases or decreases the value of each table entry in inverse proportion to the requested quality. The complete quantization tables actually used are recorded in the compressed file so that the decompressor will know how to (approximately) reconstruct the DCT coefficients.

Selection of an appropriate quantization table is something of a black art. Most existing compressors start from a sample table developed by the ISO JPEG committee. It is likely that future research will yield better tables that provide more compression for the same perceived image quality. Implementation of improved tables should not cause any compatibility problems, because decompressors merely read the tables from the compressed file; they don't care how the table was picked.

Encode the resulting coefficients

The resulting coefficients contain a significant amount of redundant data. Huffman compression will losslessly remove the redundancies, resulting in smaller JPEG data. An optional extension to the JPEG specification allows

Ex.1007 / Page 227 of 1158
APPLE INC.

arithmetic encoding to be used instead of Huffman for an even greater compression ratio. (See the section called "JPEG Extensions (Part 1)" below.) At this point, the JPEG data stream is ready to be transmitted across a communications channel or encapsulated inside an image file format.

JPEG Extensions (Part 1)

What we have examined thus far is only the baseline specification for JPEG. A number of extensions have been defined in Part 1 of the JPEG specification that provide progressive image buildup, improved compression ratios using arithmetic encoding, and a lossless compression scheme. These features are beyond the needs of most JPEG implementations and have therefore been defined as "not required to be supported" extensions to the JPEG standard.

Progressive image buildup

Progressive image buildup is an extension for use in applications that need to receive JPEG data streams and display them on the fly. A baseline JPEG image can be displayed only after all of the image data has been received and decoded. But some applications require that the image be displayed after only some of the data is received. Using a conventional compression method, this means displaying the first few scan lines of the image as it is decoded. In this case, even if the scan lines were interlaced, you would need at least 50 percent of the image data to get a good clue as to the content of the image. The progressive buildup extension of JPEG offers a better solution.

Progressive buildup allows an image to be sent in layers rather than scan lines. But instead of transmitting each bitplane or color channel in sequence (which wouldn't be very useful), a succession of images built up from approximations of the original image are sent. The first scan provides a low-accuracy representation of the entire image—in effect, a very low-quality JPEG compressed image. Subsequent scans gradually refine the image by increasing the effective quality factor. If the data is displayed on the fly, you would first see a crude, but recognizable, rendering of the whole image. This would appear very quickly because only a small amount of data would need to be transmitted to produce it. Each subsequent scan would improve the displayed image's quality one block at a time.

A limitation of progressive JPEG is that each scan takes essentially a full JPEG decompression cycle to display. Therefore, with typical data transmission rates, a very fast JPEG decoder (probably specialized hardware) would be needed to make effective use of progressive transmission.

Ex.1007 / Page 228 of 1158
APPLE INC.

A related JPEG extension provides for hierarchical storage of the same image at multiple resolutions. For example, an image might be stored at 250×250, 500×500, 1000×1000, and 2000×2000 pixels, so that the same image file could support display on low-resolution screens, medium-resolution laser printers, and high-resolution imagesetters. The higher-resolution images are stored as differences from the lower-resolution ones, so they need less space than they would need if they were stored independently. This is not the same as a progressive series, because each image is available in its own right at the full desired quality.

Arithmetic encoding

The baseline JPEG standard defines Huffman compression as the final step in the encoding process. A JPEG extension replaces the Huffman engine with a binary arithmetic entropy encoder. The use of an arithmetic coder reduces the resulting size of the JPEG data by a further 10 percent to 15 percent over the results that would be achieved by the Huffman coder. With no change in resulting image quality, this gain could be of importance in implementations where enormous quantities of JPEG images are archived.

Arithmetic encoding has several drawbacks:

- Not all JPEG decoders support arithmetic decoding. Baseline JPEG decoders are required to support only the Huffman algorithm.
- The arithmetic algorithm is slower in both encoding and decoding than Huffman.
- The arithmetic coder used by JPEG (called a *Q-coder*) is owned by IBM and AT&T. (Mitsubishi also holds patents on arithmetic coding.) You must obtain a license from the appropriate vendors if their Q-coders are to be used as the back end of your JPEG implementation.

Lossless JPEG compression

A question that commonly arises is "At what Q factor does JPEG become lossless?" The answer is "never." Baseline JPEG is a lossy method of compression regardless of adjustments you may make in the parameters. In fact, DCT-based encoders are always lossy, because roundoff errors are inevitable in the color conversion and DCT steps. You can suppress deliberate information loss in the downsampling and quantization steps, but you still won't get an exact recreation of the original bits. Further, this minimum-loss setting is a very inefficient way to use lossy JPEG.

Ex.1007 / Page 229 of 1158
APPLE INC.

The JPEG standard does offer a separate lossless mode. This mode has nothing in common with the regular DCT-based algorithms, and it is currently implemented only in a few commercial applications. JPEG lossless is a form of Predictive Lossless Coding using a 2D Differential Pulse Code Modulation (DPCM) scheme. The basic premise is that the value of a pixel is combined with the values of up to three neighboring pixels to form a predictor value. The predictor value is then subtracted from the original pixel value. When the entire bitmap has been processed, the resulting predictors are compressed using either the Huffman or the binary arithmetic entropy encoding methods described in the JPEG standard.

Lossless JPEG works on images with 2 to 16 bits per pixel, but performs best on images with 6 or more bits per pixel. For such images, the typical compression ratio achieved is 2:1. For image data with fewer bits per pixels, other compression schemes do perform better.

JPEG Extensions (Part 3)

The following JPEG extensions are described in Part 3 of the JPEG specification.

Variable quantization

Variable quantization is an enhancement available to the quantization procedure of DCT-based processes. This enhancement may be used with any of the DCT-based processes defined by JPEG with the exception of the baseline process.

The process of quantization used in JPEG quantizes each of the 64 DCT coefficients using a corresponding value from a quantization table. Quantization values may be redefined prior to the start of a scan but must not be changed once they are within a scan of the compressed data stream.

Variable quantization allows the scaling of quantization values within the compressed data stream. At the start of each 8×8 block is a quantizer scale factor used to scale the quantization table values within an image component and to match these values with the AC coefficients stored in the compressed data. Quantization values may then be located and changed as needed.

Variable quantization allows the characteristics of an image to be changed to control the quality of the output based on a given model. The variable quantizer can constantly adjust during decoding to provide optimal output.

The amount of output data can also be decreased or increased by raising or lowering the quantizer scale factor. The maximum size of the resulting JPEG file or data stream may be imposed by constant adaptive adjustments made by the variable quantizer.

Ex.1007 / Page 230 of 1158
APPLE INC.

The variable quantization extension also allows JPEG to store image data originally encoded using a variable quantization scheme, such as MPEG. For MPEG data to be accurately transcoded into another format, the other format must support variable quantization to maintain a high compression ratio. This extension allows JPEG to support a data stream originally derived from a variably quantized source, such as an MPEG I-frame.

Selective refinement

Selective refinement is used to select a region of an image for further enhancement. This enhancement improves the resolution and detail of a region of an image. JPEG supports three types of selective refinement: hierarchical, progressive, and component. Each of these refinement processes differs in its application, effectiveness, complexity, and amount of memory required.

- Hierarchical selective refinement is used only in the hierarchical mode of operation. It allows for a region of a frame to be refined by the next differential frame of a hierarchical sequence.
- Progressive selective refinement is used only in the progressive mode and adds refinement. It allows a greater bit resolution of zero and non-zero DCT coefficients in a coded region of a frame.
- Component selective refinement may be used in any mode of operation. It allows a region of a frame to contain fewer colors than are defined in the frame header.

Image tiling

Tiling is used to divide a single image into two or more smaller subimages. Tiling allows easier buffering of the image data in memory, quicker random access of the image data on disk, and the storage of images larger than 64K×64K samples in size. JPEG supports three types of tiling: simple, pyramidal, and composite.

- Simple tiling divides an image into two or more fixed-size tiles. All simple tiles are coded from left to right and from top to bottom and are contiguous and non-overlapping. All tiles must have the same number of samples and component identifiers and must be encoded using the same processes. Tiles on the bottom and right of the image may be smaller than the designated size of the image dimensions and will therefore not be a multiple of the tile size.
- Pyramidal tiling also divides the image into tiles, but each tile is also tiled using several different levels of resolution. The model of this process is the JPEG Tiled Image Pyramid (JTIP), which is a model of how to create a multi-resolution pyramidal JPEG image.

Ex.1007 / Page 231 of 1158
APPLE INC.

A JTIP image stores successive layers of the same image at different resolutions. The first image stored at the top of the pyramid is one-sixteenth of the defined screen size and is called a *vignette.* This image is used for quick displays of image contents, especially for file browsers. The next image occupies one-fourth of the screen and is called an *imagette.* This image is typically used when two or more images must be displayed at the same time on the screen. The next is a low-resolution, full-screen image, followed by successively higher-resolution images and ending with the original image.

Pyramidal tiling typically uses the process of "internal tiling," where each tile is encoded as part of the same JPEG data stream. Tiles may optionally use the process of "external tiling," where each tile is a separately encoded JPEG data stream. External tiling may allow quicker access of image data, easier application of image encryption, and enhanced compatibility with certain JPEG decoders.

- Composite tiling allows multiple-resolution versions of images to be stored and displayed as a *mosaic.* Composite tiling allows overlapping tiles that may be different sizes and have different scaling factors and compression parameters. Each tile is encoded separately and may be combined with other tiles without resampling.

SPIFF (Still Picture Interchange File Format)

SPIFF is an officially sanctioned JPEG file format that is intended to replace the defacto JFIF (JPEG File Interchange Format) format in use today. SPIFF includes all of the features of JFIF and adds quite a bit more functionality. SPIFF is designed so that properly written JFIF readers will read SPIFF-JPEG files as well.

For more information, see the article about SPIFF in Part Two of this book.

Other extensions

Other JPEG extensions include the addition of a version marker segment that stores the minimum level of functionality required to decode the JPEG data stream. Multiple version markers may be included to mark areas of the data stream that have differing minimum functionality requirements. The version marker also contains information indicating the processes and extension used to encode the JPEG data stream.

For Further Information About JPEG

The JPEG standard is available in English, French, or Spanish, and as a paper copy or a PostScript or Word for Windows document from the International

Ex.1007 / Page 232 of 1158
APPLE INC.

Standards Organization (ISO) or International Telecommunication Union (ITU). Copies of the standard may be ordered from:

American National Standards Institute, Inc.
Attention: Customer Service
11 West 42nd St.
New York, NY 10036 USA
Voice: 212-642-4900

The standard is published as both an ITU Recommendation and as an ISO/IEC International Standard, and is divided into three parts: Part 1 is the actual specification, Part 2 covers compliance-testing methods, and Part 3 covers extensions to the JPEG specification. Parts 1 and 2 are at International Standard status. See these documents:

"Digital Compression and Coding of Continuous-Tone Still Images, Requirements and Guidelines," Document number ITU-T T.81 or ISO/IEC 10918-1.

"Digital Compression and Coding of Continuous-Tone Still Images, Compliance testing," Document number ITU-T T.83 or ISO/IEC 10918-2.

Part 3 is still at Committee Draft status. See this document:

"Digital Compression and Coding of Continuous-Tone Still Images, Extensions," Document number ITU-T T.84 or ISO/IEC 10918-3.

New information on JPEG and related algorithms is constantly appearing. The majority of the commercial work for JPEG is being carried out at these companies:

Eastman Kodak Corporation
232 State Street
Rochester, NY 14650
Voice: 800-242-2424
WWW: *http://www.kodak.com*

C-Cube Microsystems
1778 McCarthy Boulevard
Milpitas, CA 95035
Voice: 408-944-6300

See the article about the JFIF file format supported by C-Cube in Part Two of this book and the SPIFF file format defined by Part 3 of the JPEG specification.

Ex.1007 / Page 233 of 1158
APPLE INC.

The JPEG FAQ (Frequently Asked Questions) is a useful source of general information about JPEG. This FAQ is included on the CD-ROM that accompanies this book; however, because the FAQ is updated frequently, the CD-ROM version should be used only for general information. The FAQ is posted every two weeks to USENET newsgroups *comp.graphics.misc*, *news.answers*, and *comp.answers*. You can get the latest version of this FAQ from the *news.answers* archive at:

ftp://rtfm.mit.edu/pub/usenet/news.answers/jpeg-faq.

You can also get this FAQ by sending email to:

mail-server@rtfm.mit.edu

with the message "send usenet/news.answers/jpeg-faq" in the body.

A consortium of programmers, the Independent JPEG Group (IJG), has produced a public domain version of a JPEG encoder and decoder in C source code form. We have included this code on the CD-ROM that accompanies this book. You can obtain the IJG library from various FTP sites, information services, and computer bulletin boards.

The best short technical introduction to the JPEG compression algorithm is:

Wallace, Gregory K., "The JPEG Still Picture Compression Standard," *Communications of the ACM*, vol. 34, no. 4, April 1991, pp. 30–44.

A more complete explanation of JPEG can be found in the following texts:

Pennebaker, William B. and Joan L. Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York, 1993.

This book contains the complete text of the ISO JPEG standards (DIS 10918-1 and 10918-2). This is by far the most complete exposition of JPEG in existence and is highly recommended.

Nelson, Mark, *The Data Compression Book*, M&T Books, Redwood City, CA. 1991.

This book provides good explanations and example C code for a multitude of compression methods, including JPEG. It is an excellent source if you are comfortable reading C code but don't know much about data compression in general. The book's JPEG sample code is incomplete and not very robust, but the book is a good tutorial.

Here is a short bibliography of additional JPEG reading:

Barda, J.F., "Codage et Compression des Grandes Images," *Proceedings of AFNOR Multimedia and Standardization Conference*, vol. 1, March 1993, pp. 300–315.

Ex.1007 / Page 234 of 1158
APPLE INC.

Hudson, G., H. Yasuda, and I. Sebestyen, "The International Standardization of Still Picture Compression Technique," *Proceedings of the IEEE Global Telecommunications Conference*, November 1988, pp. 1016–1021.

Leger, A., J. Mitchell, and Y. Yamazaki, "Still Picture Compression Algorithms Evaluated for International Standardization," *Proceedings of the IEEE Global Telecommunications Conference*, November 1988, pp. 1028–1032.

Leger, A., T. Omachi, and T.K. Wallace, "JPEG Still Picture Compression Algorithm," *Optical Engineering*, vol. 30, no. 7, July 1991, pp. 947–954.

Mitchell, J.L., and W.B. Pennebaker, "Evolving JPEG Color Data Compression Standard," in *Standards for Electronic Imaging Systems*, Neir, M. and M.E. Courtot, eds., vol. CR37, SPIE Press, 1991, pp. 68–97.

Netravali, A.N., and B.G. Haskell, *Digital Pictures: Representation and Compression*, Plenum Press, New York, 1988.

Rabbani, M., and Jones, P., *Digital Image Compression Techniques, Tutorial Texts in Optical Engineering*, vol. TT7, SPIE Press, 1991.

JBIG Compression

JBIG is a method for compressing bi-level (two-color) image data. The acronym JBIG stands for Joint Bi-level Image Experts Group, a standards committee that had its origins within the International Standards Organization (ISO). The compression standard they developed bears the name of this committee.

In 1988, the ISO and CCITT formed JBIG by joining the ISO/IEC JTC1/SC29/WG9 group and the CCITT SGVIII subgroup for the joint purpose of developing a standard, lossless method of compressing bi-level data. In 1993, the standard defining the JBIG method of bi-level data encoding was finalized and released.

The main features of JBIG are:

- Lossless compression of one-bit-per-pixel image data
- Ability to encode individual bitplanes of multiple-bit pixels
- Progressive or sequential encoding of image data

JBIG is intended to completely replace the less efficient MR (Modified READ) and MMR (Modified Modified READ) compression algorithms used by the CCITT Group 3 (G3) and Group 4 (G4) data transmission protocols,

Ex.1007 / Page 235 of 1158
APPLE INC.

respectively. In 1995, the International Telecommunication Union (ITU) proposed an extension of the G3 and G4 standards to allow the use of JBIG-compressed image data in conjunction with these protocols. (Refer to the section on CCITT compression for more details about the G3 and G4 protocols.)

On scanned images of line art and printed text, JBIG achieves compression ratios 10 percent to 50 percent greater than that of G4, and up to 500 percent greater on computer-generated images of printed text. Bi-level images processed with half-toning or dithering are compressed 2 to 30 times smaller than when compressed with G4.

JBIG achieves these impressive compression ratios by adapting to the information content of the image data being encoded. An adaptive arithmetic coder is used to predict and code future data symbols based on the characteristics of the data currently being encoded. G3 and G4 however, are non-adaptive and use the same fixed patterns and algorithms to encode all image data regardless of the content.

JBIG also supports both sequential and progressive encoding methods. Sequential encoding reads data from the top to bottom and from left to right of an image and encodes it as a single image. Progressive encoding allows a series of multiple-resolution versions of the same image data to be stored within a single JBIG data stream. In contrast, G3 and G4 only support sequential coding at a fixed resolution.

JBIG is platform-independent and implements easily over a wide variety of distributed environments. It achieves excellent compression ratios on bi-level images, and it is capable of efficiently encoding some types of color and gray-scale images as well. JBIG's progressive encoding capabilities appear to make it the obvious choice for transmitting and storing bi-level information on networked environments, such as the World Wide Web. So why isn't JBIG more popular and widespread?

It is questionable how much success JBIG will have in replacing either G3 or G4. G3 and MR are the most widely used protocol and compression method, respectively, for facsimile transmission, and they are already supported by most telecommunications equipment that use bi-level image data. And G4, while typically requiring too great a bandwidth for conventional facsimile purposes, is a primary method of data compression in most document imaging systems. G4 achieves effective compression ratios of up to 20-to-1 on both scanned and computer-generated bi-level document image data.

Ex.1007 / Page 236 of 1158
APPLE INC.

Perhaps the greatest advantage the CCITT protocols offer over JBIG is that they are free and unencumbered by patents and legal disputes. Anyone may freely implement and distribute G3 and G4 codecs without the need of licensing agreements or royalty payments. JBIG, on the other hand, contains many patented processes (24 are listed in the JBIG Recommendation); the most prominent is the IBM arithmetic Q-coder, which is an option in JPEG, but is mandatory in JBIG.

JBIG Basics

Bi-level images contain only two colors and are stored using a single bit per pixel. Black-and-white images, such as the pages of a book, are the most common type of bi-level images. However, any two colors may be represented by the 1 (foreground color) or 0 (background color) state of a single bit.

Typical bi-level compression algorithms encode only a single scan line at a time using a run-length encoding technique. Such algorithms are referred to as 1D encoding methods. 2D encoding methods encode runs of pixels by describing the differences between the pixel values in the current and the previous scan lines.

JBIG encodes redundant image data by comparing a pixel in a scan line with a set of pixels already scanned by the encoder. These additional pixels are called a *template*, and they form a simple map of the pattern of pixels that surround the pixel that is being encoded. The values of these pixels are used to identify redundant patterns in the image data. These patterns are then compressed using an adaptive arithmetic compression coder.

The adaptive nature of templates allows the color of the pixel values being encoded to be predicted with a high degree of success. For gray-scale images with halftoning, compression ratios are increased by as much as 80 percent over non-adaptive methods.

Although designed primarily as a method for compressing bi-level image data, JBIG is capable of compressing color or gray-scale images with a depth of up to 255 bits per pixel. Such multi-bit pixel images are compressed by bitplane rather than by pixel. For example, an 8-bit image compressed using JBIG would be encoded into eight separate bitplanes.

This type of encoding may be used as an alternative to lossless JPEG. JBIG has been found to produce better compression results than lossless JPEG (using the Q-coder) on images with two to five bits per pixel and to produce identical results on image data with pixels six to eight bits in depth.

Ex.1007 / Page 237 of 1158
APPLE INC.

It is recommended that each bitplane be preprocessed with a gray-coding algorithm to normalize the changes between adjacent byte values in the image data. This process increases the efficiency of the JBIG encoder.

JBIG images may be encoded sequentially or progressively. Sequentially encoded images are stored in a single layer at full resolution and without other lower resolution images being stored in the same data stream. This sequential JBIG image is equivalent in function and application to a G4 image. Such an image is decoded in a single pass and has at least as good a compression ratio as G4.

Progressively encoded images start with the highest resolution image and end with the lowest. The high-resolution image is stored in a separate layer and is then used to produce a lower resolution image, also stored in its own layer. Each layer after the first layer is called a *resolution doubling.* An image with three layers is said to have two doublings.

There is no imposed limit to the number of doublings that may be encoded. For example, a 1200-dpi image may be encoded as one layer (1200 dpi), three layers (1200, 600, and 300 dpi), or five layers (1200, 600, 300, 150, and 75 dpi). The lowest resolution is determined by whatever is considered useful. Even a 10-dpi image, though not legible, is still useful as an icon.

Progressive decoding is the opposite process, with the lowest resolution image being decoded first, followed by increased resolutions of the image until the full resolution is achieved. This technique has the advantage of allowing data to appear immediately on the output device. Only data up to the appropriate resolution of the output device need be decoded and sent.

Both sequential and progressive JBIG encoding are completely compatible. Images compressed using sequential encoding are readable by progressive JBIG decoders. Sequential JBIG decoders are only capable of reading the first, lowest-resolution layer within a progressively-encoded JBIG image.

Many applications that utilize JBIG may only have use for sequential encoding and decoding, especially those used for facsimile transmission. It is therefore possible to implement a simplified JBIG algorithm that encodes only the first layer in a JBIG data stream. Such encoders produce a valid JBIG-encoded data stream that is readable by all JBIG decoders.

Progressive encoding does not add much more data to a JBIG data stream than does sequential encoding, but it does have greater memory requirements. Because a lower resolution image is encoded from data of the next higher resolution image (and vice versa when decoding), a frame buffer must be used to store image data that is being used as a reference.

Ex.1007 / Page 238 of 1158
APPLE INC.

For Further Information About JBIG

JBIG is published both as an ITU Recommendation and as an ISO/IEC Standard:

> ITU-T Recommendation T.82|ISO/IEC 11544:1993, Coded representation of Picture and Audio Information—Progressive Bi-Level Image Compression.

This document is available, at cost, from the ITU, ISO, and many document services. For more information on the ITU and ISO, please visit their Web sites:

> *http://www.itu.ch*
> *http://www.iso.ch*

The November 1988 issue of the *IBM Journal of Research and Development* contains a set of five articles describing the IBM Q-coder and a basic, bi-level image encoder that implements the Q-coder. See:

> Pennebaker, W.B., J.L. Mitchell, G.G. Langdon, and R.B. Arps, "An Overview of the Basic Principles of the Q-coder Adaptive Binary Arithmetic Coder," *IBM Journal of Research and Development*, vol. 32, no. 6, November 1988, pp. 717–726.

The Q-coder is also described as an extension in Part 1 of the JPEG ITU-T Recommendation T.81 and ISO/IEC Standard 10918-1.

The following paper covers T.4, T.6, JBIG, and other facsimile communications protocols:

> Urban, Stephen J., "Review of Standards for Electronic Imaging for Facsimile Systems," *Journal of Electronic Imaging*, vol. 1, no. 1, January 1992, pp. 5–21.

JBIG-KIT, a JBIG bi-level image compression toolkit, is provided on the CD-ROM and also available via anonymous FTP from either of the following:

> *ftp://ftp.informatik.uni-erlangen.de/pub/doc/ISO/JBIG/jbigkit-0.8.tar.gz*
> *ftp://nic.funet.fi/pub/graphics/misc/test-images/jbig.tar.gz*

JBIG-KIT is an ANSI C implementation of the JBIG encoding standard in the form of a portable library used for encoding and decoding JBIG data streams. This library is specifically designed for 32-bit (and greater) machine architectures, although 16-bit systems are also supported.

JBIG-KIT is free software under the GNU General Public License and provides complete code and documentation. A 1992 draft copy of the CCITT T.82 Recommendation for JBIG is also currently included in this distribution.

Ex.1007 / Page 239 of 1158
APPLE INC.

NOTE

Due to patent restrictions, do not assume that it is legal to use the code in JBIG-KIT, or any other JBIG implementation, unless the proper licensing agreements have been obtained.

The author of JBIG-KIT, Markus Kuhn, may be contacted at:

mskuhn@cip.informatik.uni-erlangen.de
http://wwwcip.informatik.uni-erlangen.de/user/mskuhn

Information and discussions on JBIG may also be found on USENET in the *comp.compression.research* newsgroup and in the Frequently Asked Questions (FAQ) listing for *comp.compression*, found in the *news.answers*, *comp.answers*, and *comp.compression* newsgroups and on the CD-ROM that accompanies this book.

ART Compression

ART is a proprietary compression scheme designed and promoted by Johnson-Grace, a software development firm founded in 1992 by Steve Johnson and Chris Grace. Johnson-Grace develops software tools such as Web browsers for online services and end users.

ART Basics

As with JPEG, the degree of compression in ART is adjustable, and higher compression ratios are lossy. There is also a lossless mode. Johnson-Grace is marketing ART as a multi-purpose compressor to the online market and expects to adapt it to support audio, animation, and full-motion video in the future. As such, it will compete directly with codecs such as Intel's Indeo and the encoder/decoder packages from Cinepak and Microsoft. Johnson-Grace also plans to support the interleaving of text, graphics, audio, and video in the future.

Johnson-Grace claims that ART provides compression ratios that are typically three times smaller than either JPEG or GIF. This would, of course, be a boon to the online community if decompression were comparable to that of GIF, for example. Documentation from Johnson-Grace suggests that America Online's TurboWeb browser supports ART-compressed images.

Although the details of the algorithm are kept secret, Johnson-Grace has released some descriptive information. The algorithm seeks to analyze an image and identify a number of key features, such as color, noise, edges, and repetitive features, which are then prioritized by the relative contribution of

Ex.1007 / Page 240 of 1158
APPLE INC.

each feature to the quality of the image. The prioritizing engine uses what Johnson-Grace calls *fuzzy logic* to classify and prioritize features of the image that is being compressed. Repetitive features are identified and linked in the image using a proprietary method. Image components are quantized, and low-priority features are ignored. As with JPEG, the degree of information loss increases with the degree of compression and is offset by the degree of redundancy for a particular compression ratio.

ART apparently uses a variety of known compression methods, including wavelet compression, to optimize compression of data. Presumably, the compression algorithm used is matched to the pixel depth of the image being compressed, because Johnson-Grace claims to compete with both GIF (256-color) and JPEG (24-bit color). Johnson-Grace states that ART-compressed images are typically less than 10K in size, which enhances what they call "speed-to-screen."

ART-compressed images may be layered, which means that they can be transmitted in stages over low-bandwidth modem lines, and can provide nearly immediate, though low-quality, display on the client's display device. The display quality then improves as the rest of the information is received and is progressively rendered.

Images are compressed with an ART toolkit called ART Press (MAC ART Tools on the Macintosh). Johnson-Grace was distributing the ART toolkit free of charge until the end of 1995. Pricing had not been set as of the time of this writing.

Technologically, the ART compression scheme represents a step forward in intelligence and bodes well for the future of compression. Superior results are obtained by matching the appropriate compression technology to the image being compressed. It remains to be seen whether Johnson-Grace will be able to popularize their system in their targeted online community.

For Further Information About ART

For more information, contact:

Johnson-Grace
2 Corporate Plaza
Newport Beach CA 92660-7929
Voice: 714-759-0700 x245
FAX: 714-729-4643

Ex.1007 / Page 241 of 1158
APPLE INC.

Johnson-Grace also has a Web site, and you can find more information at:

http://www.jgc.com/cpi/art/
http://www.jgc.com/cpi/art/artcomp.html
http://www.jgc.com/freedemo/

Fractal Image Compression

Fractal encoding is a mathematical process used to encode bitmaps containing a real-world image as a set of mathematical data that describes the fractal properties of the image. Fractal encoding relies on the fact that all natural, and most artificial, objects contain redundant information in the form of similar, repeating patterns called *fractals.*

Fractal Basics

A fractal is a structure that is made up of similar forms and patterns that occur in many different sizes. The term *fractal* was first used by Benoit Mandelbrot to describe repeating patterns that he observed occurring in many different structures. These patterns appeared nearly identical in form at any size and occurred naturally in all things. Mandelbrot also discovered that these fractals could be described in mathematical terms and could be created using very small and finite algorithms and data.

Let's look at a real-world example. If you look at the surface of an object such as the floor currently beneath your feet, you will notice that there are many repeating patterns in its texture. The floor's surface may be wood, concrete, tile, carpet, or even dirt, but it still contains repeating patterns ranging in size from very small to very large.

If we make a copy of a small part of the floor's surface and compare it to every other part of the floor, we would find several areas that are nearly identical in appearance to our copy. If we change the copy slightly by scaling, rotating, or mirroring it, we can make it match even more parts of the floor. Once a match is found, we can then create a mathematical description of our copy, including any alterations we have made, and can store it, and the location of all of the parts of the floor it matches, as data.

If we repeat this process for the entire floor, we will end up with a set of mathematical equations called *fractal codes* that describe the entire surface of the floor in terms of its fractal properties. These mathematical equations can then be used to recreate an image of the entire floor.

Ex.1007 / Page 242 of 1158
APPLE INC.

The process illustrated in this example is very similar in concept to vector (2D) and 3D graphics, where mathematical descriptions of objects, rather than actual pictures of the objects themselves, are stored. The important difference between vector and fractal graphics is that fractal descriptions are derived from actual ecofactual patterns found in real-world objects, while vector and 3D objects are purely artificially generated structures that do not naturally contain fractal patterns.

Fractal encoding is largely used to convert bitmap images to fractal codes. Fractal decoding is just the reverse, in which a set of fractal codes are converted to a bitmap.

The encoding process is extremely computationally intensive. Millions or billions of iterations are required to find the fractal patterns in an image. Depending upon the resolution and contents of the input bitmap data, and output quality, compression time, and file size parameters selected, compressing a single image could take anywhere from a few seconds to a few hours (or more) on even a very fast computer.

Decoding a fractal image is a much simpler process. The hard work was performed finding all the fractals during the encoding process. All the decoding process needs to do is to interpret the fractal codes and translate them into a bitmap image.

Currently, the most popular method of fractal encoding is a process called the Fractal Transform created in 1988 by Michael F. Barnsley of Iterated Systems. Barnsley's transform was the first practical algorithm used to mathematically describe a real-world bitmap image in terms of its fractal properties.

Two tremendous benefits are immediately realized by converting conventional bitmap images to fractal data. The first is the ability to scale any fractal image up or down in size without the introduction of image artifacts or a loss in detail that occurs in bitmap images. This process of "fractal zooming" is independent of the resolution of the original bitmap image, and the zooming is limited only by the amount of available memory in the computer.

The second benefit is the fact that the size of the physical data used to store fractal codes is much smaller than the size of the original bitmap data. If fact, it is not uncommon for fractal images to be more than 100 times smaller than their bitmap sources. It is this aspect of fractal technology, called *fractal compression*, that has promoted the greatest interest within the computer imaging industry.

Ex.1007 / Page 243 of 1158
APPLE INC.

Fractal compression is lossy. The process of matching fractals does not involve looking for exact matches, but instead looking for "best fit" matches based on the compression parameters (encoding time, image quality, and size of output). But the encoding process can be controlled to the point where the image is "visually lossless." That is, you shouldn't be able to notice where the data was lost.

Fractal compression differs from other lossy compression methods, such as JPEG, in a number of ways. JPEG achieves compression by discarding image data that is not required for the human eye to perceive the image. The resulting data is then further compressed using a lossless method of compression. To achieve greater compression ratios, more image data must be discarded, resulting in a poorer quality image with a pixelized (blocky) appearance.

Fractal images are not based on a map of pixels, nor is the encoding weighted to the visual characteristics of the human eye. Instead, bitmap data is discarded when it is required to create a best-fit fractal pattern. Greater compression ratios are achieved using greater computationally intensive transforms that may degrade the image, but the distortion appears much more natural due to the fractal components.

Most other lossy methods are also symmetrical in nature. That is, a particular sequence of steps is used to compress an image, and the reverse of those steps is used to decompress it. Compression and decompression will take about the same amount of time as well. Fractal compression is an asymmetrical process, taking much longer to compress an image than to decompress it. This characteristic limits the usefulness of fractally compressed data to applications where image data is constantly decompressed but never recompressed. Fractal compression is therefore highly suited for use in image databases and CD-ROM applications.

The content and resolution of the source bitmap can greatly affect fractal compression. Images with a high fractal content (e.g., faces, landscapes, and intricate textures) result in much higher compression ratios than images with a low fractal content (e.g., charts, diagrams, text, and flat textures). High-resolution images may be compressed to achieve higher compression ratios and will still retain a high image quality. To retain a high quality for lower resolution images, the resulting compression ratio will be much lower. Images with a greater bit depth (such as 24-bit truecolor images) will also compress more efficiently than images with fewer bits per pixel (such as 8-bit gray-scale images).

The process of fractal compression is by no means in the public domain. There are many patents claiming a method of image data compression based on fractal transforms. Also, the exact process used by some fractal packages—including Barnsley's Fractal Transform—is considered proprietary.

Ex.1007 / Page 244 of 1158
APPLE INC.

For Further Information About Fractal Compression

A wealth of information on fractal compression is available both on the World Wide Web and in your local technical bookstore. On the Web the following sites will provide you with quite a bit of reading material and with links to other sites containing fractal information:

http://links.waterloo.ca/
Waterloo fractal compression page

http://inls.ucsd.edu/y/Fractals/
Yuval Fisher's fractal image compression page

http://www-rocq.inria.fr/fractales/
Groupe Fractales

http://spanky.triumf.ca/
The Spanky Fractal Database

ftp://ftp.informatik.uni-freiburg.de/documents/papers/fractal/
Dietmar Supe's FTP site for fractal papers

Fractal software packages on the Internet include:

http://www.iyu.fi/~kuru/fractalCompression/
RGB fractal compression tools

http://nic.funet.fi/pub/graphics/packages/fractal/fractal-2.0.tar
Yuval Fisher's fractal decompression code

Information on fractal encoding can also be found on the USENET newsgroups *sci.fractals, comp.compression* and *sci.math.*

Yuval Fisher's paper "Fractal Image Compression," SIGGRAPH '92 Course Notes, is a very good introduction to fractal compression and is available as a PostScript document at:

ftp://nic.funet.fi/pub/graphics/packages/fractal-image-compression/

There are many books on fractals and fractal compression. Probably the best for getting a programmer up to speed with fractal technology is:

Yuval Fisher, ed., *Fractal Image Compression: Theory and Application to Digital Images*, Springer Verlag, New York, 1995.

This book is a collection of article about fractal encoding that includes a non-mathematical introduction to fractal compression, description of modems for the encoding and decoding process, and C source code for a fractal encoder and decoder.

Ex.1007 / Page 245 of 1158
APPLE INC.

The following books also contain very good overviews of fractal technology:

Barnsley, Michael F., *Fractals Everywhere*, second edition, Academic Press, San Diego, 1993.

Barnsley, Michael F., *The Desktop Fractal Design System Versions 1 and 2*, second edition, Academic Press, San Diego, 1992.

Barnsley, Michael F., and Lyman P. Hurd, *Fractal Image Compression*, AK Peters Limited, 1993.

You might also want to look at the original Mandelbrot text:

Mandelbrot, Benoit B., *The Fractal Geometry of Nature*, WH Freeman & Co., New York, 1983.

The following journal articles are also recommended:

Anson, L.A., "Fractal Image Compression," *Byte Magazine*, October 1993.

Sloane, M.F and A.D., "A Better Way to Compress Images," *Byte Magazine*, January 1988.

McGregor, D.R., R.J. Fryer, P. Cockshott, and P. Murray, "Fast Fractal Compression," *Dr. Dobb's Journal*, vol. 21, no. 243, January 1996.

Iterated Systems produces the Images Incorporated fractal imaging application. This application supports the encoding and decoding of fractal image from over 20 different bitmap file formats. Iterated also sells a library of fractal routines you can incorporate directly into your own programs. You can contact Iterated Systems at:

Iterated Systems, Inc.
3525 Piedmont Road
Seven Piedmont Center, Suite 600
Atlanta GA 30305-1530
Voice: 800-437-2285
Fax: 404-264-8300 (sales)
Email: *support@iterated.com* (technical support)
WWW: *http://www.iterated.com/*

Two documents to read are the Iterated System's Fractal FAQ at:

http://www.iterated.com/nfaq.htm

and the Welcome to Fractals and Imaging document at:

http://www.iterated.com/nwelbook.htm

Ex.1007 / Page 246 of 1158
APPLE INC.

The Iterated Systems' Web site also provides much useful information, including Iterated's patent claims on fractal technology and its Fractal Image File (FIF) format.

For Further Information About Data Compression

On USENET the *comp.compression* newsgroup FAQ article provides a useful source of information about a variety of different compression methods. Also included is information on many archiving programs (*pkzip, lzh, zoo,* etc.) and patents pertaining to compression algorithms. This FAQ is included on the CD-ROM that accompanies this book; however, because the FAQ is updated frequently, the CD-ROM version should be used only for general information. You can get the latest version of this FAQ from the *news.answers* newsgroup at:

ftp://rtfm.mit.edu/pub/usenet/comp.compression/compression-faq

where it is kept in three parts: *part1, part2,* and *part3.* This FAQ may also be obtained by sending email to:

mail-server@rtfm.mit.edu

with the message

```
send usenet/comp.compression/compression-faq/part1
send usenet/comp.compression/compression-faq/part2
send usenet/comp.compression/compression-faq/part3
```

in the body.

There are many books on data encoding and compression. Most older books contain only mathematical descriptions of encoding algorithms. Some newer books, however, have picked up the highly desirable trend of including working (we hope) C code examples in their text and even making the code examples available online or on disk.

The following references contain good general discussions of many of the compression algorithms discussed in this chapter:

Bell, T.C, I.H. Witten, and J.G. Cleary, "Modeling for Text Compression," *ACM Computing Surveys,* vol. 21, no. 4, December 1989, p. 557.

Bell, T.C, I.H. Witten, and J.G. Cleary, *Text Compression,* Prentice-Hall, Englewood Cliffs, N.J, 1990.

Huffman, D.A., "A Method for the Construction of Minimum-Redundancy Codes," *Communications of the ACM,* vol. 40, no. 9, September 1952, pp. 1098–1101.

Ex.1007 / Page 247 of 1158
APPLE INC.

Jain, Anil K., Paul M. Farrelle, and V. Ralph Angazi, *Image Data Compression: Digital Image Processing Techniques*, Academic Press, San Diego, CA, 1984, pp. 171–226.

Lelewer, D.A, and D.S. Hirschberg, "Data Compression," *ACM Computing Surveys*, vol. 19, no. 3, September 1987, p. 261.

Storer, James A., *Data Compression: Methods and Theory*, Computer Science Press, Rockville, MD, 1988.

Storer, James A., ed., *Image and Text Compression*, Kluwer Books, Boston, MA, 1992.

Williams, R., *Adaptive Data Compression*, Kluwer Books, Boston, MA, 1990, pp. 30–44.

Ex.1007 / Page 248 of 1158
APPLE INC.

CHAPTER 10

Multimedia

Most of this book describes image file formats and the types of data compression they employ. However, still images are not the only type of data that can be stored in a file. This chapter describes the other types of graphics data that are becoming popular.

Beyond Traditional Graphics File Formats

A hot topic in the world of personal computers today is multimedia. Multimedia applications combine text, graphics, audio, and video in much the same way a motion picture film combines sound and motion photography. But, unlike motion pictures, multimedia can be interactive through the use of a keyboard, mouse, joystick, or other input device to control the behavior of the multimedia presentation. The output from a multimedia application can be through conventional speakers or a stereo system, a music or voice synthesizer, or other types of output devices.

A conventional stereo system or television and video tape recorder (VCR) are *passive information devices.* You can raise and lower the volume of a stereo, change the color of a television picture, or fast-forward a VCR, but this type of control is very limited in capability and is used only intermittently. When you use a passive information device, you normally just sit and watch the picture and listen to the sound.

Anyone who has played a computer or video arcade game has experienced an *active information device.* The games at your local video arcade, or hooked up to your living room television (and therefore permanently attached to your eight-year-old's hands), require constant input in order to function properly. And, although the sights and sounds of such a game might be staggering, the control and utility a user gains from an active information device is only slightly more than is gained using a passive one.

Ex.1007 / Page 249 of 1158
APPLE INC.

Personal computers are not only active information devices, but also interactive devices. A computer itself does very little unless a user interacts with it. Computers are, as you would expect, excellent platforms for interactive multimedia applications.

Interactive multimedia provides more than just the stimulus-response reaction of a video game. It also allows a collection of complex data to be manipulated with a much finer control than is possible using non-interactive devices. Sample multimedia applications in existence today include:

- Online, multimedia dictionaries and encyclopedias containing text, sounds, and images. These allow the instant lookup of word definitions and provide video playback, in addition to pictures.
- Games that respond to hand movements and that talk back to the user

Computerized multimedia is still in its infancy. It is currently a tool used for educational and entertainment purposes and is expanding out into the commercial world. There probably isn't a complex computerized control system that wouldn't be easier to learn or to use if it had a standardized, multimedia front end. And one day you might even see multimedia applications with heuristic algorithms that will allow your computer to learn as much from you as you will from your computer.

Multimedia File Formats

Multimedia data and information must be stored in a disk file using formats similar to image file formats. Multimedia formats, however, are much more complex than most other file formats because of the wide variety of data they must store. Such data includes text, image data, audio and video data, computer animations, and other forms of binary data, such as Musical Instrument Digital Interface (MIDI), control information, and graphical fonts. (See the "MIDI Standard" section later in this chapter.) Typical multimedia formats do not define new methods for storing these types of data. Instead, they offer the ability to store data in one or more existing data formats that are already in general use.

For example, a multimedia format may allow text to be stored as PostScript or Rich Text Format (RTF) data rather than in conventional ASCII plain-text format. Still-image bitmap data may be stored as BMP or TIFF files rather than as raw bitmaps. Similarly, audio, video, and animation data can be stored using industry-recognized formats specified as being supported by that multimedia file format.

Ex.1007 / Page 250 of 1158
APPLE INC.

Multimedia formats are also optimized for the types of data they store and the format of the medium on which they are stored. Multimedia information is commonly stored on CD-ROM. Unlike conventional disk files, CD-ROMs are limited in the amount of information they can store. A multimedia format must therefore make the best use of available data storage techniques to efficiently store data on the CD-ROM medium.

There are many types of CD-ROM devices and standards that may be used by multimedia applications. If you are interested in multimedia, you should become familiar with them.

The original Compact Disc first introduced in early 1980s was used for storing only audio information using the CD-DA (Compact Disc-Digital Audio) standard produced by Phillips and Sony. CD-DA (also called the Red Book) is an optical data storage format that allows the storage of up to 74 minutes of audio (764 megabytes of data) on a conventional CD-ROM.

The CD-DA standard evolved into the CD-XA (Compact Disc-Extended Architecture) standard, or what we call the CD-ROM (Compact Disc-Read Only Memory). CD-XA (also called the Yellow Book) allows the storage of both digital audio and data on a CD-ROM. Audio may be combined with data, such as text, graphics, and video, so that it may all be read at the same time. An ISO 9660 file system may also be encoded on a CD-ROM, allowing its files to be read by a wide variety of different computer system platforms.

The CD-I (Compact Disc-Interactive) standard defines the storage of interactive multimedia data. CD-I (also called the Green Book) describes a computer system with audio and video playback capabilities designed specifically for the consumer market. CD-I units allow the integration of fully interactive multimedia applications into home computer systems.

A still-evolving standard is CD-R (Compact Disc-Recordable or Compact Disc-Write Once), which specifies a CD-ROM that may be written to by a personal desktop computer and read by any CD-ROM player.

For more specific information on multimedia, refer to the articles on the RIFF, DVI, QuickTime, and MPEG multimedia formats in Part Two of this book.

Types of Data

The following sections describe various types of data that you might find, in addition to static graphics data, in multimedia files.

Ex.1007 / Page 251 of 1158
APPLE INC.

Animation

Somewhere between the motionless world of still images and the real-time world of video images lies the flip-book world of computer animation. All of the animated sequences seen in educational programs, motion CAD renderings, and computer games are computer-animated (and in many cases, computer-generated) animation sequences.

Traditional cartoon animation is little more than a series of artwork cells, each containing a slight positional variation of the animated subjects. When a large number of these cells is displayed in sequence and at a fast rate, the animated figures appear to the human eye to move.

A computer-animated sequence works in exactly the same manner. A series of images is created of a subject; each image contains a slightly different perspective on the animated subject. When these images are displayed (played back) in the proper sequence and at the proper speed (frame rate), the subject appears to move.

Computerized animation is actually a combination of both still and motion imaging. Each frame, or cell, of an animation is a still image that requires compression and storage. An animation file, however, must store the data for hundreds or thousands of animation frames and must also provide the information necessary to play back the frames using the proper display mode and frame rate.

Animation file formats are only capable of storing still images and not actual video information. It is possible, however, for most multimedia formats to contain animation information, because animation is actually a much easier type of data than video to store.

The image-compression schemes used in animation files are also usually much simpler than most of those used in video compression. Most animation files use a delta compression scheme, which is a form of Run-Length Encoding that stores and compresses only the information that is different between two images (rather than compressing each image frame entirely). RLE is relatively easy to decompress on the fly. (See Chapter 9, *Data Compression*, for a description of RLE compression.)

Storing animations using a multimedia format also produces the benefit of adding sound to the animation (what's a cartoon without sound?). Most animation formats cannot store sound directly in their files and must rely on storing the sound in a separate disk file which is read by the application that is playing back the animation.

Ex.1007 / Page 252 of 1158
APPLE INC.

Animations are not only for entertaining kids and adults. Animated sequences are used by CAD programmers to rotate 3D objects so they can be observed from different perspectives; mathematical data collected by an aircraft or satellite may be rendered into an animated fly-by sequence. Movie special effects benefit greatly by computer animation.

For more specific information on animation, refer to the articles on the FLI and GRASP animation formats in Part Two of this book.

Digital Video

One step beyond animation is broadcast video. Your television and video tape recorder are a lot more complex than an 8mm home movie projector and your kitchen wall. There are many complex signals and complicated standards that are involved in transmitting those late-night reruns across the airwaves and cable. Only in the last few years has a personal computer been able to work with video data at all.

Video data normally occurs as continuous, analog signals. In order for a computer to process this video data, we must convert the analog signals to a non-continuous, digital format. In a digital format, the video data can be stored as a series of bits on a hard disk or in computer memory.

The process of converting a video signal to a digital bitstream is called analog-to-digital conversion (A/D conversion), or digitizing. A/D conversion occurs in two steps:

1. Sampling captures data from the video stream.
2. Quantizing converts each captured sample into a digital format.

Each sample captured from the video stream is typically stored as a 16-bit integer. The rate at which samples are collected is called the *sampling rate.* The sampling rate is measured in the number of samples captured per second (samples/second). For digital video, it is necessary to capture millions of samples per second.

Quantizing converts the level of a video signal sample into a discrete, binary value. This value approximates the level of the original video signal sample. The value is selected by comparing the video sample to a series of predefined threshold values. The value of the threshold closest to the amplitude of the sampled signal is used as the digital value.

A video signal contains several different components which are mixed together in the same signal. This type of signal is called a composite video signal and is

Ex.1007 / Page 253 of 1158
APPLE INC.

not really useful in high-quality computer video. Therefore, a standard composite video signal is usually separated into its basic components before it is digitized.

The composite video signal format defined by the NTSC (National Television Standards Committee) color television system is used in the United States. The PAL (Phase Alternation Line) and SECAM (Sequential Coleur Avec Memoire) color television systems are used in Europe and are not compatible with NTSC. Most computer video equipment supports one or more of these system standards.

The components of a composite video signal are normally decoded into three separate signals representing the three channels of a color space model, such as RGB, YUV, or YIQ. Although the RGB model is quite commonly used in still imaging, the YUV, YIQ, or YCbCr models are more often used in motion-video imaging. TV practice uses YUV or similar color models because the U and V channels can be downsampled to reduce data volume without materially degrading image quality.

The three composite channels mentioned here are the same channels used in the downsampling stage of JPEG compression; for more information, see the section called "JPEG Compression" of Chapter 9.

Once the video signal is converted to a digital format, the resulting values can be represented on a display device as pixels. Each pixel is a spot of color on the video display, and the pixels are arranged in rows and columns just as in a bitmap. Unlike a static bitmap, however, the pixels in a video image are constantly being updated for changes in intensity and color. This updating is called scanning, and it occurs 60 times per second in NTSC video signals (50 times per second for PAL and SECAM).

A video sequence is displayed as a series of frames. Each frame is a snapshot of a moment in time of the motion-video data, and is very similar to a still image. When the frames are played back in sequence on a display device, a rendering of the original video data is created. In real-time video the playback rate is 30 frames per second. This is the minimum rate necessary for the human eye to successfully blend each video frame together into a continuous, smoothly moving image.

A single frame of video data can be quite large in size. A video frame with a resolution of 512 x 482 will contain 246,784 pixels. If each pixel contains 24 bits of color information, the frame will require 740,352 bytes of memory or disk space to store. Assuming there are 30 frames per second for real-time video, a

Ex.1007 / Page 254 of 1158
APPLE INC.

10-second video sequence would be more than 222 megabytes in size! It is clear there can be no computer video without at least one efficient method of video data compression.

There are many encoding methods available that will compress video data. The majority of these methods involve the use of a transform coding scheme, usually employing a Fourier or Discrete Cosine Transform (DCT). These transforms physically reduce the size of the video data by selectively throwing away unneeded parts of the digitized information. Transform compression schemes usually discard 10 percent to 25 percent or more of the original video data, depending largely on the content of the video data and upon what image quality is considered acceptable.

Usually a transform is performed on an individual video frame. The transform itself does not produce compressed data. It discards only data not used by the human eye. The transformed data, called coefficients, must have compression applied to reduce the size of the data even further. Each frame of data may be compressed using a Huffman or arithmetic encoding algorithm, or even a more complex compression scheme such as JPEG. (See Chapter 9 for a discussion of these compression methods.) This type of intraframe encoding usually results in compression ratios between 20:1 to 40:1 depending on the data in the frame. However, even higher compression ratios may result if, rather than looking at single frames as if they were still images, we look at multiple frames as temporal images.

In a typical video sequence, very little data changes from frame to frame. If we encode only the pixels that change between frames, the amount of data required to store a single video frame drops significantly. This type of compression is known as interframe delta compression, or in the case of video, motion compensation. Typical motion compensation schemes that encode only frame deltas (data that has changed between frames) can, depending on the data, achieve compression ratios upwards of 200:1.

This is only one possible type of video compression method. There are many other types of video compression schemes, some of which are similar and some of which are different. For more information on compression methods, refer to Chapter 9 and to the articles in Part Two that describe multimedia file formats.

Digital Audio

All multimedia file formats are capable, by definition, of storing sound information. Sound data, like graphics and video data, has its own special

Ex.1007 / Page 255 of 1158
APPLE INC.

requirements when it is being read, written, interpreted, and compressed. Before looking at how sound is stored in a multimedia format we must look at how sound itself is stored as digital data.

All of the sounds that we hear occur in the form of analog signals. An analog audio recording system, such as a conventional tape recorder, captures the entire sound wave form and stores it in analog format on a medium such as magnetic tape.

Because computers are now digital devices it is necessary to store sound information in a digitized format that computers can readily use. A digital audio recording system does not record the entire wave form as analog systems do (the exception being Digital Audio Tape [DAT] systems). Instead, a digital recorder captures a wave form at specific intervals, called the sampling rate. Each captured wave-form snapshot is converted to a binary integer value and is then stored on magnetic tape or disk.

Storing audio as digital samples is known as Pulse Code Modulation (PCM). PCM is a simple quantizing or digitizing (audio to digital conversion) algorithm, which linearly converts all analog signals to digital samples. This process is commonly used on all audio CD-ROMs.

Differential Pulse Code Modulation (DPCM) is an audio encoding scheme that quantizes the difference between samples rather than the samples themselves. Because the differences are easily represented by values smaller than those of the samples themselves, fewer bits may be used to encode the same sound (for example, the difference between two 16-bit samples may only be four bits in size). For this reason, DPCM is also considered an audio compression scheme.

One other audio compression scheme, which uses difference quantization, is Adaptive Differential Pulse Code Modulation (ADPCM). DPCM is a non-adaptive algorithm. That is, it does not change the way it encodes data based on the content of the data. DPCM uses the sample number of bits to represent every signal level. ADPCM, however, is an adaptive algorithm and changes its encoding scheme based on the data it is encoding. ADPCM specifically adapts by using fewer bits to represent lower-level signals than it does to represent higher-level signals. Many of the most commonly used audio compression schemes are based on ADPCM.

Digital audio data is simply a binary representation of a sound. This data can be written to a binary file using an audio file format for permanent storage much in the same way bitmap data is preserved in an image file format. The data can be read by a software application, can be sent as data to a hardware device, and can even be stored as a CD-ROM.

Ex.1007 / Page 256 of 1158
APPLE INC.

The quality of an audio sample is determined by comparing it to the original sound from which it was sampled. The more identical the sample is to the original sound, the higher the quality of the sample. This is similar to comparing an image to the original document or photograph from which it was scanned.

The quality of audio data is determined by three parameters:

- Sample resolution
- Sampling rate
- Number of audio channels sampled

The *sample resolution* is determined by the number of bits per sample. The larger the sampling size, the higher the quality of the sample. Just as the apparent quality (resolution) of an image is reduced by storing fewer bits of data per pixel, so is the quality of a digital audio recording reduced by storing fewer bits per sample. Typical sampling sizes are eight bits and 16 bits.

The *sampling rate* is the number of times per second the analog wave form was read to collect data. The higher the sampling rate, the greater the quality of the audio. A high sampling rate collects more data per second than a lower sampling rate, therefore requiring more memory and disk space to store. Common sampling rates are 44.100 kHz (higher quality), 22.254 kHz (medium quality), and 11.025 kHz (lower quality). Sampling rates are usually measured in the signal processing terms hertz (Hz) or kilohertz (kHz), but the term samples per second (samples/second) is more appropriate for this type of measurement.

A sound source may be sampled using one channel (monaural sampling) or two channels (stereo sampling). Two-channel sampling provides greater quality than mono sampling and, as you might have guessed, produces twice as much data by doubling the number of samples captured. Sampling one channel for one second at 11,000 samples/second produces 11,000 samples. Sampling two channels at the same rate, however, produces 22,000 samples/second.

The amount of binary data produced by sampling even a few seconds of audio is quite large. Ten seconds of data sampled at low quality (one channel, 8-bit sample resolution, 11.025 samples/second sampling rate) produces about 108K of data (88.2 Kbits/second). Adding a second channel doubles the amount of data to produce nearly a 215K file (176 Kbits/second). If we increase the sample resolution to 16 bits, the size of the data doubles again to 430K (352 Kbits/second). If we now increase the sampling rate to 22.05 Ksamples/second, the amount of data produced doubles again to 860K (705.6

Ex.1007 / Page 257 of 1158
APPLE INC.

Kbits/second). At the highest quality generally used (two channels, 16-bit sample resolution, 44.1 Ksamples/second sampling rate), our 10 seconds of audio now requires 1.72 megabytes (1411.2 Kbits/second) of disk space to store.

Consider how little information can really be stored in 10 seconds of sound. The typical musical song is at least three minutes in length. Music videos are from five to 15 minutes in length. A typical television program is 30 to 60 minutes in length. Movie videos can be three hours or more in length. We're talking a lot of disk space here.

One solution to the massive storage requirements of high-quality audio data is data compression. For example, the CD-DA (Compact Disc-Digital Audio) standard performs mono or stereo sampling using a sample resolution of 16 bits and a sampling rate of 44.1 samples/second, making it a very high-quality format for both music and language applications. Storing five minutes of CD-DA information requires approximately 25 megabytes of disk space—only half the amount of space that would be required if the audio data were uncompressed.

Audio data, in common with most binary data, contains a fair amount of redundancy that can be removed with data compression. Conventional compression methods used in many archiving programs (zoo and pkzip, for example) and image file formats don't do a very good job of compressing audio data (typically 10 percent to 20 percent). This is because audio data is organized very differently from either the ASCII or binary data normally handled by these types of algorithms.

Audio compression algorithms, like image compression algorithms, can be categorized as lossy and lossless. Lossless compression methods do not discard any data. The decompression step produces exactly the same data as was read by the compression step. A simple form of lossless audio compression is to Huffman-encode the differences between each successive 8-bit sample. Huffman encoding is a lossless compression algorithm and, therefore the audio data is preserved in its entirety.

Lossy compression schemes discard data based on the perceptions of the psychoacoustic system of the human brain. Parts of sounds that the ear cannot hear, or the brain does not care about, can be discarded as useless data.

An algorithm must be careful when discarding audio data. The ear is very sensitive to changes in sound. The eye is very forgiving about dropping a video frame here or reducing the number of colors there. The ear, however, notices even slight changes in sounds, especially when specifically trained to recognize audial infidelities and discrepancies. However, the higher the quality of an

Ex.1007 / Page 258 of 1158
APPLE INC.

audio sample, the more data will be required to store it. As with lossy image compression schemes, at times you'll need to make a subjective decision between quality and data size.

Audio

There is currently no "audio file interchange format" that is widely used in the computer-audio industry. Such a format would allow a wide variety of audio data to be easily written, read, and transported between different hardware platforms and operating systems.

Most existing audio file formats, however, are very machine-specific and do not lend themselves to interchange very well. Several multimedia formats are capable of encapsulating a wide variety of audio formats, but do not describe any new audio data format in themselves.

Many audio file formats have headers just as image files do. Their header information includes parameters particular to audio data, including sample rate, number of channels, sample resolution, type of compression, and so on. An identification field ("magic" number) is also included in several audio file format headers.

Several formats contain only raw audio data and no file header. Any parameters these formats use are fixed in value and therefore would be redundant to store in a file header. Stream-oriented formats contain packets (chunks) of information embedded at strategic points within the raw audio data itself. Such formats are very platform-dependent and would require an audio file format reader or converter to have prior knowledge of just what these parameter values are.

Most audio file formats may be identified by their file types or extensions. Some common sound file formats are:

.AU	Sun Microsystems
.SND	NeXT
HCOM	Apple Macintosh
.VOC	SoundBlaster
.WAV	Microsoft Waveform
AIFF	Apple/SGI
8SVX	Apple/SGI

A multimedia format may choose to either define its own internal audio data format or simply encapsulate an existing audio file format. Microsoft

Ex.1007 / Page 259 of 1158
APPLE INC.

Waveform files are RIFF files with a single Waveform audio file component, while Apple QuickTime files contain their own audio data structures unique to QuickTime files. For further information about audio, see the section called "Audio Formats" in Chapter 1, *Introduction.*

MIDI Standard

Musical Instrument Digital Interface (MIDI) is an industry standard for representing sound in a binary format. MIDI is not an audio format, however. It does not store actual digitally sampled sounds. Instead, MIDI stores a description of sounds, in much the same way that a vector image format stores a description of an image and not image data itself.

Sound in MIDI data is stored as a series of control messages. Each message describes a sound event using terms such as pitch, duration, and volume. When these control messages are sent to a MIDI-compatible device (the MIDI standard also defines the interconnecting hardware used by MIDI devices and the communications protocol used to interchange the control information) the information in the message is interpreted and reproduced by the device.

MIDI data may be compressed, just like any other binary data, and does not require special compression algorithms in the way that audio data does.

For Further Information

Information about multimedia products from Microsoft may be obtained from the following address:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

The following documents, many of which are included in the Microsoft Multimedia Development Kit (MDK), contain information on multimedia applications and file formats:

Microsoft Windows Multimedia Development Kit (MDK) 1.0 Programmers Reference

Microsoft Windows 3.1 Software Development Kit (SDK) Multimedia Programmer's Reference

Microsoft Windows Multimedia Programmer's Guide

Microsoft Windows Multimedia Programmer's Reference

Ex.1007 / Page 260 of 1158
APPLE INC.

Multimedia Developer Registration Kit (MDRK)

Multimedia Programming Interface and Data Specification 1.0, August 1991

Microsoft Multimedia Standards Update March 13, 1993, 2.0.0

A great deal of useful information about multimedia files and applications may be found at the following FTP site:

ftp://ftp.microsoft.com/developer/drg/Multimedia/

See these Web pages for multimedia information:

http://www.microsoft.com
Microsoft homepage

http://ac.dal.ca/~dong/contents.html
Multimedia file formats on the Internet

http://vizwiz.gmd.de/MultimediaInfo/
Index to multimedia information resources

http://www.yahoo.com/Computers_and_Internet/Multimedia
Yahoo multimedia resources

The specification for MIDI may be obtained from:

International MIDI Association (IMA)
5316 West 57th Street
Los Angles, CA 90056
213-649-6434

Refer to the articles on Microsoft RIFF, Intel DVI, MPEG, and QuickTime in Part Two of this book for specific information on multimedia file formats.

Ex.1007 / Page 261 of 1158
APPLE INC.

PART TWO

Graphics File Formats

NAME:	Adobe Illustrator
ALSO KNOWN AS:	AI, Adobe AI
TYPE:	Metafile
COLORS:	Unlimited
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	ASCII
ORIGINATOR:	Adobe
PLATFORM:	Macintosh, MS Windows, NeXT
SUPPORTING APPLICATIONS:	Adobe Illustrator, most desktop publishing packages, most drawing packages, Adobe Streamline
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	CGM

USAGE: Storage and interchange of line-based artwork.

COMMENTS: A widely used format for the exchange of 2D objects. Basic files are simple to write. Unfortunately, files created by applications implementing the full AI specification can be large and complex and may be slow to render.

Overview

Originally written for the Macintosh platform, Adobe Illustrator is a well-known and widely used drawing application. There are currently Macintosh, Microsoft Windows, and NeXT versions. Much of the power of Adobe Illustrator comes from its implementation of Bezier splines as drawing objects, and the fact that it presents a simple user interface for precise positioning of spline-based drawing objects. Bezier splines have some advantages for the modeling of natural (and some human-made) objects. AI files are also used to distribute clip art.

The AI format encapsulates and formalizes a subset of the PostScript page description language (PDL) in a structured file. Such files are meant to be

Ex.1007 / Page 263 of 1158
APPLE INC.

imaged on a PostScript printer, but may include a bitmap version of the image to facilitate screen preview. PostScript is a powerful and complex language in its full implementation, and this complexity is partly due to its ability to specify almost anything that can appear on a 2D output device. AI, however, is tailored to the storage of graphics data in the conventional sense: drawings, artwork, and lettering used for ornamental and display purposes. Note that AI files can still be quite complex. PostScript derives much of its power from the ability to define sequences of operations and to later concatenate them using a simple syntax. This hidden complexity is sometimes, but not always, minimized in AI files.

Simple AI files are quite easy to construct, and an application can create files that can be read by any AI reader or can be printed on any PostScript printer. Reading AI files is another matter entirely. Certain operations may be difficult for a rendering application to implement or simulate. In light of this, developers often choose not to render the image from the PostScript-subset line data in the file. Note, however, that almost all of the image can usually be reconstructed using simple operations. If you wish to develop an AI reader, it can be done, and you can get hints by examining the source code of the GNU GhostScript system, which provides a nearly full implementation of the PostScript language.

PostScript, and consequently the AI subset, has its own language and conventions. We suggest that you read the PostScript documents prior to working with AI files or perusing the specification document included on the CD-ROM that comes with this book. These are referenced at the end of this article.

File Organization

AI files consist of a series of ASCII lines, which may be comments, data, commands, or combinations of commands and data. Commands in AI files are operators, which may or may not be followed by data. Data is pushed and popped off a stack, and operators use data on the stack in LIFO order. PostScript is sometimes thought of as a stream-oriented PDL. Lines, however, must be parsed first as full lines and then tokenized. Operator lines usually have the following form:

```
arg-list operator
```

Key concepts are path, stroke, fill, and the graphics state. A path is traveled by a graphics cursor. Stroking results in a path or portion of a path eventually being displayed on the rendering surface. Fills operate on closed paths. The

Ex.1007 / Page 264 of 1158
APPLE INC.

results of stroking and what actually happens during a fill operation (among other things) are determined by the graphics state.

Comments are any line where the first non-whitespace character is `%`. Special lines, known as *structuring comments*, are designated by the double-comment `%%`. A + immediately after a structuring comment designator indicates that the data on the line is associated with the previous structuring comment.

In the discussion that follows, llx, lly, urx, and ury refer to lower-left x, lower-left y, upper-right x, and upper-right y, respectively. These are used to denote bounding box rectangles and are similar to how rectangles were specified in the original Macintosh development environment. This can cause problems in other systems because many objects are oriented relative to their bounding boxes. Developers in other environments should consider themselves forewarned.

AI files are organized as follows: a file ID line, followed by a header, followed by the rest of the file in which the graphics objects are defined. In Adobe terminology, the header consists of structuring and other comments known as the Prolog. Following the header is a Script Setup section, which consists of the drawing commands defining the objects in the image, a section called the Page Trailer, and a section called the Document Trailer. Files are terminated with the structuring comment `%%EOF`, signaling to the rendering application or device that the data associated with the image to be rendered is complete.

ID line
Prolog
Comments
Script Setup
Objects
Page Trailer
Document Trailer

Ex.1007 / Page 265 of 1158
APPLE INC.

File Details

All AI files start with a comment line in the following format:

```
%!PS-Adobe-X.X EPSF-Y.Y
```

where *X.X* and *Y.Y* are the AI and EPSF (format or the encapsulated bitmap), respectively. Remember that your reader must examine and tokenize the first line, whether or not it is a comment, in order to identify the file.

This is followed by a series of structuring and other Adobe-defined comments and a host of procedure sets that make up the header.

```
%File ID line
%%BeginProlog
   .
   .                                          list of header comments
   .
%%EndComments
   .
   .                                          list of procedure set definitions
   .
%%EndProlog
```

A typical header appears as follows:

```
%%BeginProlog
%%Creator: Adobe Illustrator 4.0
%%For: John Doe Xylane University
%%Title: Figure 1.1
%%CreationDate: 12-13-95 03:43:15
%%BoundingBox: 0 0 512 512
%%EndComments
%%DocumentSuppliedResources: procset AdobeCustomColor Red123 3 0
%%EndProlog
```

Note that Adobe Illustrator can save files locally in a "no-header" version, in which case much of this information—except for the file ID line and the BoundingBox comment—will be absent. Files saved by Adobe Illustrator for export will generally contain more extensive header information. Adobe advises that a document in the following format is the mimimum acceptable for reading by Adobe Illustrator:

```
%!PS-Adobe-3.0
%%BoundingBox llx lly urx ury
%%EndComments
%%EndProlog
%%BeginSetup
   .
   .                                          font encoding
```


Ex.1007 / Page 266 of 1158
APPLE INC.

```
    .
    .
    .                                          pattern definitions
    .
%%EndSetup
    .
    .                                          object definition
    .
%%Trailer
%%EOF
```

Structuring Comments

Some structuring comments are used only in the header. Others appear later in the file.

The Microsoft Windows version of Adobe Illustrator (v4.0) uses the following structuring comments:

```
%%Creator: Adobe Illustrator(TM) version
%%For: user, organization
%%Title: title
%%CreationDate: date, time
%%DocumentProcSets: Adobe_Illustrator_version level revision
%%DocumentSuppliedProcsets: Adobe_Illustrator_version level revision
%%DocumentFonts: fonts
%%BoundingBox: llx lly urx ury
%%TemplateBox: llx lly urx ury
%%Template: filename
%%PageOrigin: x y
%%PrinterName: printer
%%PrinterRect: llx lly urx ury
```

These last four comments are normally stored in the resource fork in the Macintosh environment.

The following are comments used in the header. Most are optional, and some are Macintosh-specific and are ignored on other platforms.

```
%%BeginProlog
%%BoundingBox: llx lly urx ury                 Integers
%%CMYKCustomColors: custom-color-name
%%CreationDate: date time                      Valid PostScript strings
%%Creator: name version                        Program used to create file
%%DocumentCustomColors: custom-color           Color defined in file
%%DocumentFiles: file-to-import
%%DocumentFonts: font-name                     Font used in image
%%DocumentNeededResources: other-resources     Needed to reconstruct image
%%DocumentProcessColors: colors                Colors in color model definition
%%EndComments
%%EndProlog
```


Ex.1007 / Page 267 of 1158
APPLE INC.

```
%%EOF
%%For: name                          Optional info
%%IncludeResource: name              Misc necessary to reconstruct image
%%Title: name                        Name of image
```

Procedure Sets

Procedure set definitions start with the DocumentSuppliedResources keyword, and are followed by the keyword procset and a list of arguments consisting of one or more of the following:

Adobe_cmykcolor
Adobe_cshow
Adobe_customcolor
Adobe_Illustrator_AI3
Adobe_packedarray
Adobe_pattern_AI3
Adobe_typography_AI3

These are followed by the version and revision numbers, as shown in the following example:

```
DocumentSuppliedResources: procset Adobe_cshow 3 4
```

This is the case for all but Adobe_customcolor, which adds the color name and has the following syntax:

```
DocumentSuppliedResources: procset Adobe_customcolor Red123 3 4
```

Note that among the structuring comments, only BoundingBox must be included in every AI file and helps denote the size and scale of the image.

Header Comments

The following header comments may be optionally included between the BeginProlog and EndProlog statements in the header, at the end of the rest of the header comments.

```
%AI3_ColorUsage: Black&White or Color
%AI3_TemplateBox: llx lly urx ury
%AI3_TemplateFile: vol::dir id:name
%AI3_TileBox: llx lly urx ury        Mac-specific
```


Ex.1007 / Page 268 of 1158
APPLE INC.

Script Setup Section

The Script Setup section of the file consists of setup information, a list of object definitions, and the page and document trailers.

The setup portion of this section has the following format:

```
%%BeginSetup
%%IncludeFont: font
   .
   .                                          procedure set initialization
   .
   .
   .                                          font encoding
   .
   .
   .                                          pattern definitions
   .
%%EndSetup
```

The font specified on the IncludeFont line is one used in the document and should be substituted for if the correct font is unavailable to the rendering application. Procedure sets defined in the prolog often need to be initialized and are deinitialized (or terminated) later in the file. Font encoding is the process of mapping ASCII codes to glyphs found in the font file. We won't attempt to explain font encoding here, and instead we refer you to the PostScript references at the end of this article. Note, however, that the TE and TZ operators are used to specify font encoding in this section of the file.

Patterns are also defined in this section and are meant to be used one or more times in the file. Pattern definitions have the following syntax:

```
%%BeginPattern:
pattern-name llx lly urx ury layer-list E
%%EndPattern
```

Patterns are actually miniature drawings and can have the full complexity of full-sized, multi-layered images. For this reason we refer you to the format specification for further information.

Following the setup portion of the Script Setup section is the script body, consisting of object definitions. These generally form the bulk of the file and can often be parsed and rendered to give a good approximation of the image, particularly in files produced by applications other than Adobe Illustrator.

Ex.1007 / Page 269 of 1158
APPLE INC.

Preceding each object is a flag, denoted by flag A. If A is set, then the object is considered locked (non-editable) in Adobe Illustrator. Objects may be one of the following:

path	placed art
path mask	subscriber
composite	graph
text	PostScript document

In the discussion below, we only describe path objects, placed art objects, subscriber objects, and graph objects in detail. Please consult the AI format document and PostScript specification for further information on composite and text objects. Note that the graph terminology in particular only makes sense with a copy of the Adobe Illustrator application documentation in hand.

Path objects are defined by specifying paint style, path geometry, and path render.

Path paint style is specified by setting the current graphics state. Note that the graphics state can be saved temporarily and restored. The most important of the state arguments are the following:

Argument	Operator	Meaning
array phase	d	Sets dash pattern using array
flatness	i	Bezier path flattening (0–100)
flag	D	Winding order (0=clockwise)
linecap	J	0=butted, 1=round, 2=square
linejoin	j	0=mitered, 1=round, 2=beveled
linewidth	w	Width of line (minimum 0)
miterlimit	M	Adjusts mitering (>1)

Path geometry is specified using the following syntax:

Argument	Operator	Meaning
x y	m	moveto
x y	l	lineto
x y	L	lineto corner

Ex.1007 / Page 270 of 1158
APPLE INC.

Argument	Operator	Meaning
x1 y1 x2 y2 x3 y3	c	curveto—append Bezier to path
x1 y1 x2 y2 x3 y3	C	curveto corner
x2 y2 x3 y3	v	Insert Bezier starting with current point
x2 y2 x3 y3	V	Insert Bezier to corner
x1 y1 x3 y3	y	Append Bezier to current and finish with x3,y3
x1 y1 x3 y3	Y	Append Bezier to current to corner

Paths can be rendered using the following operators:

Argument	Operator	Meaning
NA	N	Leaves unclosed, unfilled, unstroked path
NA	n	Leaves closed, unfilled, unstroked path
NA	F	Fills path, leaving it unclosed
NA	f	Fills path, closing it
NA	S	Strokes path with current default values
NA	s	Closes path and strokes with current defaults
NA	B	Leaves unclosed, but fills and strokes path
NA	b	Leaves closed, filled, stroked path

Placed art objects are EPS files included in an AI file. They can be included either by reference (to an external file) or inline. Both types of included files begin with the single-quote (') operator and end with a tilde (~) operator. Between these is the %%IncludeFile: comment for externally referenced files, or a %%BeginDocument and %%EndDocument pair for inline files. The

```
a b c d tx ty llx lly urx ury filename '
```

Arguments *a*, *b*, *c*, *d*, *tx*, and *ty* are optional and specify the transformation matrix for the file to be included. Arguments *llx*, *lly*, *urx*, and *ury* specify the bounding box of the included file. Argument *filename* is the full pathname of the included file.

Externally referenced files:

```
a b c d tx ty llx lly urx ury filename '
%%IncludeFile: filename
~
```


Ex.1007 / Page 271 of 1158
APPLE INC.

Inline files:

```
a b c d tx ty llx lly urx ury filename '
%%BeginDocument: filename
.
. included file contents
.
%%EndDocument
~
```

Macintosh systems may include references to subscriber objects, associated with the the publish and subscribe feature available in System 7. These are indicated by the *%%subscriber* comment:

```
%AI3_Subscriber: res-number
placed art object
```

where *res-number* is the resource number of the SECT resource in the file, and the *placed art object* is as described above.

Graph objects are used to specify graphs for business, technical, and scientific purposes. A number of commands allow full control over labeled illustrations, including the placement and appearance of the following (Adobe-defined) parts. Many details of these commands make sense only in the Adobe Illustrator environment:

axes	legend group
label group	data column
axis tick	series 0
category axis group	series 1
edge	

Argument	Operator	Meaning
NA	Gs	Start of graph object
NA	GS	End of graph object
l t r b	Gb	Graph bound (left,top,right,bottom)
(see spec)	Gy	Values in Graph Style dialog box
(see spec)	Gd	Values in Graph Style dialog box
axis string	Ga	Axis: 1=bottom, 2=left, 3=right
(see spec)	GA	Axis specs
r c fr fc	Gz	Cell table: rows, columns, first row, first column
cv1, cv2 . . . cvx	Gc	Reads cell values 1-x into table

Ex.1007 / Page 272 of 1158
APPLE INC.

Argument	Operator	Meaning
cw1, cw2 . . . cwx num	Gw	Column widths 1-x, num=number of columns
NA	GC	Cell table is complete
NA	Gt	Start graph customizations
NA	GT	End graph customizations
target cust	Gx	Target, customization (see below)
cust	Gp	General Illustrator customization (see below)
method	G+	Change method: 0=reset to new, 1=add new to previous
direction	Gl	0=send to back, 1=send to front
df ds fcs scs m	Gf	doFill, doStroke, fillStyle, strokeStyle, isAMask (see below)
column	Gl	Column index for table
row	Gr	Row index for table
axis	Gi	Which axis object is inside: 1=bottom, 2=left, 4=right, 8=top
(see below)	Gm	Matrix customizations
(see below)	GD	Bar design customizations
repeat	Ge	Repeat bar design
tickvalue	Gv	Numeric value corresponding to customized tick mark
NA	GX	End of customization
target col row axis	Go	Type of graph object just read in (see below)

Gx graph customizations hold information about current defaults. The target parameter is one of the following:

0 Entire graph
1 All series and legend marks
2 One series and legend marks
3 One series but no legend
4 One data bar, line, or wedge
5 All data marks
6 One series and legend marks
7 One series but no legend marks

Ex.1007 / Page 273 of 1158
APPLE INC.

8 One data line segment's marks
9 One axis, complete
10 Category axis main line
11 One axis major tick mark
12 One axis single major tick mark
13 One axis set of tick labels
14 One axis single tick label
15 All legend text
16 One legend text
17 One numerical axis main line
18 One legend's box or line but no mark
19 One legend's mark
20 All labels along one category axis
21 One label of category axis
22 Entire shadow event
23 Every tick of one axis
24 All minor ticks one axis
25 One minor tick one axis

The customization argument is one of the following:

0 Illustrator Customization
1 Set Series Graph Style
2 Set Column (Bar) Design
3 Set Mark Design

Gp customizations may be one of the following:

0 Move/Shear/Rotate/Scale
1 Set Paint Style
9 Send to Front/Back
11 Set Layout Style

The Gf operator allows the setting of Set Paint Style customization. The doFill, doStroke, and isAMask operators are flags. Operators fillStyle and strokeStyle may be:

Ex.1007 / Page 274 of 1158
APPLE INC.

0 Black (or white)
1 Process
2 Pattern
3 Custom color
4 Blend (AI for Windows v4.0)

The Gm operator specifies matrix customizations. Arguments are in the form:

```
a b c d h v generalGraphType reserved1 reserved2
```

where *a, b, c, d, h,* and *v* are matrix values. Argument *generalGraphType* specifies the type of graph to apply the customization to:

1 Grouped-column and stacked-column
2 Scatter and line graphs
3 Pie charts
4 Area graphs
5 All graphs

Arguments *reserved1* and *reserved2* are set to 0.

The GD operator has the following form:

```
designName designType repeatPartialType rotateLegend GD
```

where *designName* is a name (string), and *designType* can be:

6 Vertically scaled design
7 Uniformly scaled design
8 Repeating design
9 Sliding design

Argument *rotateLegend* is a flag that when set indicates the legend in the design box is rotated. Argument *repeatPartialType* can be:

16 Chop partial values
17 Scale partial values

Ex.1007 / Page 275 of 1158
APPLE INC.

The target argument that is associated with operator Go can have the following values:

1	All series with legends
2	One series, including legends
4	One data bar, line, or wedge
5	All data marks
6	One series and its legend marks
8	One data line segment's marks
9	One axis, including text, ticks, line
10	Category axis main line
15	All legend text
20	All labels along category axis
22	Entire shadow object

Script Trailer

The Script Trailer has the following syntax:

```
%%Trailer
   .
   .                                          procedure set termination
   .
```

The procedure set terminations section consists of an explicit list of procedure set termination commands, in the reverse order from those specified in the script setup section. An example of a procedure set termination is as follows:

```
Adobe_customcolor /terminate get exec
```

For Further Information

Much of the information for this article came from the following document:

> Adobe Illustrator File Format Specification, Adobe Developer Support Version 3.0 (Draft), 28 October 1992.

The specification is included on the CD-ROM. This is described as a draft, even though it has been relatively stable since 1989.

Ex.1007 / Page 276 of 1158
APPLE INC.

You may also contact Adobe at the following address:

Adobe Systems
Developer Support
1585 Charleston Rd.
P.O. Box 7900
Mountain View CA 94039-7900
Voice: 415-961-4400
Voice: 800-344-8335
FAX: 415-961-3769

Adobe also provides a great deal of information online, and is to be congratulated for their willingness to aid developers:

http://www.adobe.com/
ftp://ftp.adobe.com/

Adobe's PostScript documentation is widely available, and is required reading for anyone working with AI files, and is a model of coherence and lucidity; it was published in a set of books in conjunction with Addison-Wesley. Among the books are a tutorial on PostScript programming and the Type 1 font specification. They are available at most computer bookstores. The most important document for understanding the AI format is:

PostScript Language Reference Manual, second edition, Addison-Wesley, Reading, MA. (ISBN 0-201-18127-4)

Ex.1007 / Page 277 of 1158
APPLE INC.

Adobe Photoshop

NAME:	Adobe Photoshop 3.0
ALSO KNOWN AS:	PSD, Adobe PSD, Photoshop 3.0
TYPE:	Bitmap
COLORS:	Unlimited
COMPRESSION:	Uncompressed, RLE
MAXIMUM IMAGE SIZE:	30,000×30,000
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Adobe
PLATFORM:	Microsoft Windows, Apple Macintosh
SUPPORTING APPLICATIONS:	Adobe Photoshop, Adobe Premiere, desktop publishing programs
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	MacPaint, TIFF

USAGE: Storage of images altered or manipulated in the Adobe Photoshop environment.

COMMENTS: A flexible format that is easily read and written but lacks a superior compression scheme. It provides specially good support for different color storage schemes.

Overview

Adobe's Photoshop is probably the fullest featured and most highly respected commercial image-processing bitmap manipulation program in the PC and Macintosh worlds. Its wide distribution has meant that image data is often left in PSD format files and may persist in this form after the original image data is long gone.

Photoshop users are often professionals working with images in truecolor. Images can be very large, and Photoshop users are expected to have adequate memory to load and manipulate large files. Adobe has chosen to optimize the speed of reading and writing in the Photoshop system, and this is reflected in the RLE compression incorporated in the PSD format. Files, therefore, are not

Ex.1007 / Page 278 of 1158
APPLE INC.

as small as they might be using another compression method. Although this makes sense in the context of a Photoshop-equipped workstation, the lack of a superior compression scheme has probably prevented PSD from becoming more popular as a general-purpose interchange format. Because PSD is an application-specific format, expect it to change in the future.

Under Microsoft Windows, Photoshop files are stored with the PSD suffix and can be identified by looking for the file ID value 8BPS. On the Macintosh, Photoshop files are resource-fork only, and data is recognized by the file ID value 8BPS.

Earlier versions of the format had no compression and were tied to the Macintosh platform.

File Organization

PSD files consist of a header and three informational sections, called the Mode Block, Image Resources Block, and Layer and Mask Information Block, respectively. These are followed by the actual image data. The header is a series of fixed fields, and the other blocks are variable-length.

Header
Color Mode Data Block
Image Resources Block
Layer and Mask Information Block
Image Data

File Details

This section describes the details of the various sections of a PSD file.

Header

The header is 26 bytes in length and is structured as follows:

```
typedef struct _PSD_HEADER
{
   BYTE Signature[4];    /* File ID "8BPS" */
   WORD Version;         /* Version number, always 1 */
   BYTE Reserved[6];     /* Reserved, must be zeroed */
```


Ex.1007 / Page 279 of 1158
APPLE INC.

```
    WORD Channels;        /* Number of color channels (1-24) including alpha
                             channels */
    LONG Rows;            /* Height of image in pixels (1-30000) */
    LONG Columns;         /* Width of image in pixels (1-30000) */
    WORD Depth;           /* Number of bits per channel (1, 8, and 16) */
    WORD Mode;            /* Color mode */
} PSD_HEADER;
```

Only Channels values of 1–16 are supported in v2.5 files. Also in v2.5, only Depth values of 1 and 8 bits per channel are supported.

Data in the mode field specifies how the image is to be interpreted.

Mode	Description
0	Bitmap (monochrome)
1	Gray-scale
2	Indexed color (palette color)
3	RGB color
4	CMYK color
7	Multichannel color
8	Duotone (halftone)
9	Lab color

Color Mode Data Block

Following the header is the Color Mode Data block. At the start of this block is a LONG number, in big-endian format, specifying the length of the block in bytes. The nature of the following data is related to the value of the mode field of the header. If the mode field value is anything other than 2 or 6, the length of the block will be zero, and no data will follow the 4-byte length field. If the mode field value is 2, however, signifying indexed color, the following 768 bytes will contain a 256-color palette. If the mode field value is 6, signifying duotone data, the data following presumably consists of screen parameters and other related information. Unfortunately, it is intentionally not documented by Adobe, and non-Photoshop readers are advised to treat duotone images as gray-scale images.

Image Resources Block

Following the Color Mode Data block is the Image Resources block. Like the Color Mode Data block, the first four bytes are a LONG number in big-endian format specifying the length of the block. The following data consists of non-pixel data associated with an image. Information in the Image Resources block

Ex.1007 / Page 280 of 1158
APPLE INC.

was stored in the resource fork in early Photoshop versions running on the Macintosh platform.

```
struct _ColorModeDataBlock
{
    BYTE Type[4];   /* Always "8BIM" */
    WORD ID;        /* (See table below) */
    BYTE Name[];    /* Even-length Pascal-format string, 2 bytes or longer */
    LONG Size;      /* Length of resource data following, in bytes */
    BYTE Data[];    /* Resource data, padded to even length */
};
```

The format of the data is determined by the value in the ID field, which can have the following values:

ID	Data Format	Description
03e8	WORD[5]	Channels, rows, columns, depth, and mode
03e9		Optional Macintosh print manager information
03eb		Indexed color table
03ed	(See below)	Resolution information
03ee	BYTE[]	Alpha channel names (Pascal-format strings)
03ef	(See below)	Display information for each channel
03f0	BYTE[]	Optional Pascal-format caption string
03f1	LONG, WORD	Fixed-point border width, border units (see below)
03f2		Background color
03f3	BYTE[8]	Print flags (see below)
03f4		Gray-scale and halftoning information
03f5		Color halftoning information
03f6		Duotone halftoning information
03f7		Gray-scale and multichannel transfer function
03f8		Color transfer functions
03f9		Duotone transfer functions
03fa		Duotone image information
03fb	BYTE[2]	Effective black and white value for dot range
03fc		
03fd		EPS options
03fe	WORD, BYTE	Quick Mask channel ID, flag for mask initially empty
03ff		

Ex.1007 / Page 281 of 1158
APPLE INC.

Adobe Photoshop (cont'd)

ID	Data Format	Description
0400	WORD	Index of target layer (0=bottom)
0401		Working path
0402	WORD[]	Layers group info, group ID for dragging groups
0403		
0404		IPTC-NAA record
0405		Image mode for raw-format files
0406		JPEG quality (Adobe internal)
07d0–0bb6		Saved path information
0bb7		Clipping pathname
2710	(See below)	Print flags information

ID values 03e8, 03eb, 03ff, and 0403 are considered obsolete. Values 03e8 and 03eb are associated with Photoshop v2.0. The data format for values 03f2, 03f4–03fa, 03fc, 03fd, 0405–0bb7 is intentionally not documented by Adobe, or the data is missing. Please refer to the Adobe Photoshop SDK for information on obtaining the IPTC-NAA record 2 structure definition.

ID value 03ed indicates that the data is in the form of a ResolutionInfo structure:

```
typedef struct _ResolutionInfo
{
   LONG hRes;              /* Fixed-point number: pixels per inch */
   WORD hResUnit;          /* 1=pixels per inch, 2=pixels per centimeter */
   WORD WidthUnit;         /* 1=in, 2=cm, 3=pt, 4=picas, 5=columns */
   LONG vRes;              /* Fixed-point number: pixels per inch */
   WORD vResUnit;          /* 1=pixels per inch, 2=pixels per centimeter */
   WORD HeightUnit;        /* 1=in, 2=cm, 3=pt, 4=picas, 5=columns */
} RESOLUTIONINFO;
```

ID value 03ef indicates that the data is stored as a DisplayInfo structure, which contains display information associated with each channel:

```
typedef _DisplayInfo
{
   WORD  ColorSpace;
   WORD  Color[4];
   WORD  Opacity;          /* 0-100 */
   BYTE  Kind;             /* 0=selected, 1=protected */
   BYTE  Padding;          /* Always zero */
} DISPLAYINFO;
```


Ex.1007 / Page 282 of 1158
APPLE INC.

ID value 03f3 indicates that the data is a series of eight flags, indicating the enabled state of labels, crop marks, color bars, registration marks, negative, flip, interpolate, and caption items in the Photoshop Page Setup dialog box.

ID value 2710 signals that the Data section contains a WORD-length version number (should be 1), a BYTE-length flag indicating crop marks, a BYTE-length field (should be 0), a LONG-length bleed width value, and a WORD indicating the bleed width scale.

Layer and Mask Information Block

Following the Image Resources block is the Layer and Mask Information block, structured like the Color Mode Data and Image Resources blocks. In PSD files produced by Photoshop v2.5 the 4-byte Layer and Mask Information block length field contains a LONG number set to zero, as will vers3.0 files containing no layer or mask information. PSD file writers are advised that this is an area likely to be altered or expanded in future versions of the PSD format. Layer information in this bock is stored first, followed by mask information.

Layer Information

The Layer section of the Layer and Mask Information block starts with a LONG value specifying the length of the Layer Info section. This is followed by a WORD value count of the number of Layer Records to follow. Layer Records follow in sequence; a description of their organization is found below. Note that Layer Records vary in size from file to file depending on the number of channels in the image. There is one Layer Record for each layer in the image.

Data Structure	Field Name	Description
LONG	Top	Rectangle bounding the layer
LONG	Left	
LONG	Bottom	
LONG	Right	
WORD	Channels	Number of channels in the layer

The next area following contains a series of Channel Length Info records, defined as follows:

```
typedef struct _CLI
{
  WORD  ChannelID;            /* Channel Length Info field one */
  LONG  LengthOfChannelData;  /* Channel Length Info field two */
} CLI;
```


Ex.1007 / Page 283 of 1158
APPLE INC.

Adobe Photoshop (cont'd)

Data Structure	Field Name	Description
CLI	Channel Length Info #1	
CLI	Channel Length Info #2	
⋮		
CLI	Channel Length Info #*n*	
BYTE[4]	Blend Mode Signature	Always "8BIM"
BYTE[4]	Blend Mode Key	(See table below)
BYTE	Opacity	0–255 (transparent to opaque)
BYTE	Clipping	0=base, 1=non-base
BYTE	Flags	Bit 0 = transparency protected, bit 1 = visible
BYTE	Padding	Set to zero
LONG	ExtraDataSize;	

The following area is the Layer Mask data section. If there is no Layer Mask, Size (the first LONG value) is set to zero, and the Layer Mask data section is four bytes long. If the value is non-zero, the Layer Mask data section is 24 bytes in length.

Data Structure	Field Name	Description
LONG	Size	
LONG	Top	Rectangle bounding layer mask
LONG	Left	
LONG	Bottom	
LONG	Right	
BYTE	DefaultColor	0 or 255
BYTE	Flags	Bit 0=position, bit 1=layer mask disabled, bit 2=invert layer mask
WORD	padding	Set to zero

The following area contains information defining the Layer Blending Ranges. This is followed by a series of records defining the source and destination ranges for each color channel. Each source and destination field consists of two black values (0–255) followed by two white values (0–255). Each channel source and destination range record has the following format:

Ex.1007 / Page 284 of 1158
APPLE INC.

```
typedef struct _CSDR
{
  BYTE Source[4];       /* First Channel Source Range */
  BYTE Dest[4];         /* First Channel Destination Range */
} CSDR;
```

Data Structure	Field Name	Description
LONG	Size	Total length of layer blending data
BYTE[4]	Gray Blend Source	
BYTE[4]	Gray Blend Destination	
CSDR	Channel Source Range #1	
CSDR	Channel Source Range #2	
⋮		
CSDR	Channel Source Range #*n*	
BYTE[]	Layer Name	Pascal string, padded to multiple of 4 bytes

The Blend Mode Key field can contain the following ASCII keys:

Key	Meaning
norm	Normal
dark	Darken
lite	Lighten
hue	Hue
sat	Saturation
colr	Color
lum	Luminosity
mul	Multiply
scrn	Screen
diss	Dissolve
over	Overlay
hLit	Hard light
sLit	Soft light
diff	Difference

Following the list of Layer Records is the channel image data. Channel image data is stored in the form of 8-bit bitmaps. The first WORD value contains information about compression: 0 indicates that the data is uncompressed, and 1 indicates that the data is RLE compressed. Following this WORD is the actual

Ex.1007 / Page 285 of 1158
APPLE INC.

channel image data. The length of the data is equal to the number of pixels in the image, which can be calculated from the bounding box. The data may be RLE compressed, however, using the PackBits algorithm described below. If the length of the channel image is odd, a pad byte is inserted to make the end of the image land on a WORD boundary.

Mask Information
Mask information consists of one or more mask info structures having the following format:

Data Structure	Field Name	Description
WORD	Overlay Color Space	(Not documented)
BYTE[8]	Color Components	4x2 byte color components
WORD	Opacity	0=transparent, 1=opaque
BYTE	Kind	0=inverted, 1=protected, 128=use stored value
BYTE	padding	set to zero

Image Data

After the Layer and Mask Information block is a WORD containing a value acting as a compression flag. If the flag is set (to 1), the image data is RLE compressed.

The image data is stored as indicated in the header and the compression value preceding it. If the data is compressed, each line is preceded by a WORD containing a value indicating the length, in bytes, of the data associated with that line. Lines are stored in scan-line order with no padding. Multiplanar data, which may or may not be compressed, is stored as a series of planes in sequence. That is, red data is stored first, followed by green data and then blue data. If the data is multiplanar, the WORD at the start of each line is the size, in bytes, of the total data associated with each scan-line (rows-in-line times number-of-channels).

Data is compressed using the Macintosh ROM PackBit encoding scheme, which is also part of the TIFF standard. The first byte of scan-line data read is a run count. If the MSB is set, the byte is converted to its two's complement value, and the next byte read is repeated that number of times. If the MSB is zero, one is added to the count, and the next RunCount bytes are read. In pseudo-code, this is:

Ex.1007 / Page 286 of 1158
APPLE INC.

```
Zero BytesRead and Count
Read WORD LengthOfLineInBytes

Read a byte of data
Increment BytesRead
  If high bit is one
    Count is two's complement of byte
    Read next byte
    Write this value Count times
  If high bit is zero
    Count is byte value plus one
    Read next byte and write value Count times
  Increment BytesRead
  If BytesRead equals LengthOfLine the scan-line is done
```

Remember that LengthOfLineInBytes refers to all of the data associated with the current scan-line. If the image data is multiplanar, then BytesRead will have to be adjusted accordingly.

Adobe Photoshop Raw File Format

The Photoshop Raw file format is used to import data into Photoshop from applications that cannot write file formats that are recognized by Photoshop. The Raw format is also used to export data from Photoshop that can be read by such applications.

The Raw format file is an uncompressed, binary file that contains only image data and no header or color palette information. The header data must be entered into Photoshop when the file is imported.

The Raw file data is nothing more than a series of pixel values starting from the upper-left pixel in the image and continuing downward. Each value is stored as a BYTE with a value in the range 0 (black, or least intense) to 255 (white, or most intense). RGB pixels are always stored in red-green-blue order, and CMYK pixels are always stored in cyan-magenta-yellow-key order.

To import a raw file, choose the Open item from the File menu, and click Show All Files. Choose Raw from the File pop-up menu, and click Open to display the Raw dialog box. Enter the width and height of the image in pixels (1 to 30,000), number of color channels (1 for gray-scale and palette color, 3 for RGB, 4 for CMYK, and so on), and the header size (the default is 0). Click OK.

Ex.1007 / Page 287 of 1158
APPLE INC.

When you export a raw file, only the image data is saved to the file, and all of the associated header and color palette data is lost. This data must be re-entered when importing the raw file. To export a raw file, choose Save As from the File menu and Raw from the Save As pop-up menu to display the Raw dialog box. Enter the File Type and File Creator values used to identify the Raw file, the header value (the default is 0), and the Interleaved Order (interleaved is the default). Click OK.

For Further Information

The primary source of information on Photoshop is the following document, which Adobe has graciously allowed us to include on the CD-ROM that accompanies this book.

> Adobe Photoshop 3.0.4 Software Development Kit, Copyright 1991–1995, Adobe Systems

Some of the information for this article came from the following document, which is also included on the CD-ROM:

> Adobe Photoshop 2.5 Format, Copyright 1992 and 1993, by Thomas Knoll and Adobe Systems, Inc.

For additional information, contact:

> Adobe Systems
> Developer Support
> 1585 Charleston Rd.
> P.O. Box 7900
> Mountain View CA 94039-7900
> Voice: 415-961-4400
> Voice: 800-344-8335
> FAX: 415-961-3769
> WWW: *http://www.adobe.com/*
> FTP: *ftp://ftp.adobe.com/*

Ex.1007 / Page 288 of 1158
APPLE INC.

NAME:	Atari ST graphics file formats
ALSO KNOWN AS:	.ANI, .ANM, .CE1, .CE2, .CE3, .FLM, .UC1, .UC2, .UC3., NEO, .PAC, .PC1, .PC2, .PC3, PC3, .PI1, .PI2, .PI3, .RGB, .SEQ, .TNY, .TN1, .TN2, .TN3
TYPE:	Bitmap and animation
COLORS:	Typically 16
COMPRESSION:	None and RLE
MAXIMUM IMAGE SIZE:	Typically 320×200 pixels
MULTIPLE IMAGES PER FILE:	Yes (animation formats only)
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Various Atari ST software developers
PLATFORM:	Atari ST
SUPPORTING APPLICATIONS:	Many
SPECIFICATION ON CD:	Yes (third-party description)
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	IFF

USAGE: All of these formats are used by paint and animation packages found on the Atari ST.

COMMENTS: The Atari ST, with its superior graphics capabilities, was a natural platform for development of multimedia, so much of the multimedia developments of today are based on these formats.

Overview

The Atari ST computer is the home of many sparsely documented image file formats. Many of these formats are used specifically for storing animation images and dumps of images displayed on the screen. Although the Electronic Arts IFF format is used by most Atari ST paint and animation programs, many software developers have devised their own special-purpose formats to fill their needs.

Ex.1007 / Page 289 of 1158
APPLE INC.

File Organization and Details

This section contains a brief description of each of the Atari ST file formats; each format has its own file extension.

Animatic Film Format (.FLM)

The Animatic Film file format (file extension .FLM) stores a sequence of low-resolution 16-color images, which are displayed as an animation. Files in the .FLM format are stored as a header followed by one or more frames of image data. The header is 26 bytes in length and has the following format:

```
typedef struct _AnimaticFilmHeader
{
    WORD NumberOfFrames;       /* Number of frames in the animation */
    WORD Palette[16];          /* Color palette */
    WORD FilmSpeed;            /* Speed of playback */
    WORD PlayDirection;        /* Direction of play */
    WORD EndAction;            /* Action to take after last frame */
    WORD FrameWidth;           /* Width of frame in pixels */
    WORD FrameHeight;          /* Height of frame in pixels */
    WORD MajorVersionNumber;   /* Animatic major version number */
    WORD MinorVersionNumber;   /* Animatic minor version number */
    LONG MagicNumber;          /* ID number (always 27182818h) */
    LONG Reserved[3];          /* Unused (all zeros) */
} ANIMATICFILMHEADER;
```

NumberOfFrames specifies the total number of frames in the animation.

Palette is the color palette for the animation, stored as an array of 16 WORD values.

FilmSpeed is the number of delay (vblank) frames to display between each animation frame. The value of this field may be in the range 0 to 99.

PlayDirection is the direction the animation is played. Values for this field are 00h for forwards and 01h for backwards.

EndAction specifies the action to take when the last frame of the animation is reached during playback. A value of 00h indicates that the player should pause and then repeat the animation from the beginning. A value of 01h indicates that the animation should immediately repeat from the beginning (loop). A value of 03h indicates that playback should repeat in the reverse direction.

FrameWidth and FrameHeight are the size of the animation frames in pixels.

MajorVersionNumber and MinorVersionNumber contain the version number of the Animatic software that created the animation.

Ex.1007 / Page 290 of 1158
APPLE INC.

MagicNumber contains an identification value for Animatic Film files. This value is always 27182818h.

Reserved is 12 bytes of space reserved for future header fields. All bytes in this field have the value 00h.

ComputerEyes Raw Data Format (.CE1 and .CE2)

The ComputerEyes Raw Data Format is found in a low-resolution (file extension .CE1) and a medium-resolution (file extension .CE2) format. The header is 10 bytes in length and has the following format:

```
typedef struct _ComputerEyesHeader
{
    LONG Id;          /* Identification value (always 45594553h) */
    WORD Resolution;  /* Image data resolution */
    BYTE Reserved[8]; /* Miscellaneous data */
} COMPUTEREYESHEAD;
```

Id is a value used to identify a file as containing ComputerEyes-format data. The value of this field is always 45594553h or "EYES" as an ASCII string.

Resolution is the resolution of the image data stored in the file. This value is 00h for low-resolution data and 01h for high-resolution data.

Reserved is eight bytes of additional information, which is not needed for decoding the image data.

If the Resolution field value is 00h (low resolution), then the image data will be divided into three 320×220 RGB planes. Each plane is 64,000 bytes in size and is stored in red, green, blue order. The image data stores one pixel per byte, and only the lower six bits of each byte are used. Low-resolution image data is stored vertically, so rows of data are read along the Y-axis and not along the X-axis as in most other formats.

If the Resolution field value is 01h (high resolution), then the image data is stored in a single 640×480 plane, which is always 256,000 bytes in size. The image data stores one pixel per WORD, with the red value stored in bits 0 through 4, green in bits 5 through 9, and blue in bits 10 through 14. Bit 15 is not used. High-resolution image data is also stored along the vertical, rather than the horizontal, axis of the bitmap.

Ex.1007 / Page 291 of 1158
APPLE INC.

Cyber Paint Sequence Format (.SEQ)

The Cyber Paint Sequence file format (file extension .SEQ) is used for storing sequences of 16-color low-resolution images used in animations. Cyber Paint also supports an efficient form of delta-encoded data compression.

```
typedef struct _CyberPaintHeader
{
    WORD MagicNumber;                   /* Identification number */
    WORD VersionNumber;                 /* Version number */
    LONG NumberOfFrames;                /* Total number of frames */
    WORD DisplayRate;                   /* Display speed */
    BYTE Reserved[118];                 /* Unused */
    LONG FrameOffsets[NumberOfFrames];  /* Array of frame offsets */
} CUVERPAINTHEAD;
```

MagicNumber is an identification number used to indicate that the file contains Cyber Paint Sequence data. This value is typically FEDBh or FEDCh.

VersionNumber is the version number of the format.

NumberOfFrames specifies the number of data frames stored.

DisplayRate is the number of delay (vblank) frames to display between each animation frame.

Reserved is a 118-byte field reserved for future header fields. This field is set to a value of 00h.

FrameOffsets is an array of LONG offset values with a number of elements equal to the value stored in the NumberOfFrames field. Each offset value indicates the starting position of each frame, calculated from the beginning of the file.

Each frame contains a header of descriptive information in the following format:

```
typedef struct _CyberPaintFrame
{
    WORD Type;              /* Frame type */
    WORD Resolution;        /* Frame Resolution */
    WORD Palette[16];       /* Color palette */
    BYTE FileName[12];      /* Name of frame data file */
    WORD Limits;            /* Color animation limits */
    WORD Speed;             /* Color animation speed and direction */
    WORD NumberOfSteps;     /* Number of color steps */
    WORD XOffset;           /* Left position of frame on display */
    WORD YOffset;           /* Top position of frame on display */
    WORD FrameWidth;        /* Width of the frame in pixels */
    WORD FrameHeight;       /* Height of the frame in pixels */
```


Ex.1007 / Page 292 of 1158
APPLE INC.

```
    BYTE Operation;        /* Graphics operation */
    BYTE Compression;      /* Data storage method */
    LONG DataSize;         /* Length of the frame data in bytes */
    BYTE Reserved[60];     /* Unused */
} CYBERPAINTFRAME;
```

Type is an identification value identifying the header as belonging to a frame.

Resolution is the resolution of the frame data and is usually 00h.

Palette is an array of values for the 16-color palette for this frame.

FileName stores the name of the disk file in which the frame data is stored. The default string stored in this field is " . ", which indicates no filename.

Limits is the color animation limits of the frame.

Speed specifies the speed and direction of the playback.

NumberOfSteps is the number of color steps in the image data.

XOffset is the left position of the frame on the display. This value may be in the range of 0 to 319.

YOffset is the top position of the frame on the display. This value may be in the range of 0 to 199.

FrameWidth and FrameHeight are the size of the frame in pixels.

Operation is the graphics operation to perform on the frame data. A value of 00h indicates copy, and a value of 01h indicates an exclusive OR.

Compression indicates whether the frame data is compressed (a value of 01h) or uncompressed (a value of 00h).

DataSize is the actual size of the data (compressed or uncompressed) stored in the frame.

Reserved is a 60-byte field reserved for future header fields. All bytes in this field have the value 00h.

Frame data stored in a Sequence file is always 320×200 pixels. The frame data is stored as four bitplanes, with one pixel stored per WORD. Pixels are always stored along the vertical (Y) axis of the bitmap. Therefore, the first 200 bytes of frame data are the first pixels of the first bitplane of the frame, and so on.

Frame data may be compressed using a delta-encoding algorithm. Using this technique, only the changes between frames are actually encoded. Interframe data that does not change is not saved.

Ex.1007 / Page 293 of 1158
APPLE INC.

The first frame in a sequence is always stored in its entirety. You have to start someplace. Each frame thereafter is compared to the previous frame, and only the X and Y coordinates of rectangular regions of pixel values (called *change boxes*) that have changed are saved. Only one change box is stored per frame.

Each change box may be stored in one of five different variations, always using the variation that yields the best compression for a particular change box. These variations are:

- Uncompressed Copy, where the frame data is uncompressed and is simply copied onto the screen at coordinates specified in the XOffset and YOffset header fields.
- Uncompressed EOR, where the frame data is exclusive ORed with the data already at XOffset,YOffset.
- The frame data is compressed and must be uncompressed before copying to the screen.
- Compressed EOR, where the frame data must be uncompressed before it is exclusive ORed with the screen data.
- Null Frame, which contains no data (height and width are 00h) and is treated as the previous frame.

Compressed data contains a sequence of control WORDs (16-bit signed WORDs) and data. A control WORD with a value between 1 and 32,767 indicates that the next WORD is to be repeated a number of times equal to the control WORD value. A control WORD with a negative value indicates that a run of bytes equal to the absolute value of the control WORD value is to be read from the compressed data.

DEGAS Format (.PI1, .PI2, .PI3, .PC1, .PC2, .PC3)

The DEGAS animation file format actually occurs in three different variations. The DEGAS and DEGAS Elite formats support low, medium, and high-resolution graphics data (files have the extension .PI1, .PI2, and .PI3 respectively). The DEGAS Elite Compressed format supports low, medium, and high-resolution graphics data (files have the extension .PC1, .PC2, and .PC3 respectively), and it also supports data compression.

The DEGAS format stores only a single image of the display. The header is 34 bytes long and is followed by 32,000 bytes of image data:

Ex.1007 / Page 294 of 1158
APPLE INC.

```
typedef struct _DegasHeader
{
    WORD Resolution;       /* Image resolution */
    WORD Palette[16];      /* Color palette */
} DEGASHEAD;
```

Resolution is the resolution of the image data stored as a bit-field. Valid values are:

00h Left
01h Off
02h Right

Palette is an array of 16 WORD values that holds the color palette for the image.

The DEGAS Elite format contains the same header and image data structure as the DEGAS format. It differs form the DEGAS format in that it has a 32-byte footer containing additional information:

```
typedef struct _DegasEliteFooter
{
    WORD LeftColor[4];     /* Left color animation limit table */
    WORD RightColor[4];    /* Right color animation limit table */
    WORD Direction[4];     /* Animation channel direction flag */
    WORD Delay[4];         /* Animation channel delay */
} DEGASELITEFOOT;
```

LeftColor stores the left color animation limit table containing the starting color numbers for the animation.

RightColor stores the right color animation limit table containing the ending color numbers for the animation.

Direction contains the animation channel direction bit-field flag. Valid values are:

00h Left
01h Off
02h Right

Delay is the animation channel delay rate between frames. This value is measured in 1/60 of a second and is subtracted from the constant 128 to calculate this value.

The DEGAS Elite Compressed format contains the same header and footer as the DEGAS Elite format, with one variation in the header data.

Ex.1007 / Page 295 of 1158
APPLE INC.

The Resolution field uses the following bit values to indicate the resolution of the image data:

8000h Low resolution
8001h Medium resolution
8002h High resolution

The compression algorithm used is identical to RLE scheme found in the Interchange file format (IFF); see the article on the Interchange format for details.

RGB Intermediate Format (.RGB)

The RGB Intermediate Format (file extension .RGB) is actually three low-resolution DEGAS .PI1 files concatenated into a single file. The pixel data in each plane contains an actual red, green, or blue color-channel value rather than an index into the 16-color palette. On the Atari ST, there are only three bits per color channel. The Atari ST with the extended color palette uses four bits per color channel.

The structure of an entire RGB file is shown here:

```
struct _RgbFile
{
    WORD RedResolution;      /* Red plane resolution (ignored) */
    WORD RedPalette[16];     /* Red plane palette (ignored) */
    WORD RedPlane[16000];    /* Red plane data */
    WORD GreenResolution     /* Green plane resolution (ignored) */
    WORD GreenPalette        /* Green plane palette (ignored) */
    WORD GreenPlane[16000]; /* Green plane data */
    WORD BlueResolution;     /* Blue plane resolution (ignored) */
    WORD BluePalette;        /* Blue plane palette (ignored) */
    WORD BluePlane[16000];   /* Blue plane data */
}
```

Imagic Film/Picture Format (.IC1, .IC2, .IC3)

The Imagic Format stores low-, medium-, and high-resolution image data using the file extensions .IC1, .IC2, and .IC3 respectively. The header is 49 bytes long and is formatted as follows:

```
typedef struct _ImagicHeader
{
    BYTE Id[4];              /* File identification value */
    WORD Resolution;         /* Image resolution */
    WORD Palette[16];        /* Color palette */
    WORD Date;               /* Date stamp */
    WORD Time;               /* Time stamp */
```


Ex.1007 / Page 296 of 1158
APPLE INC.

```
    BYTE Name[8];           /* Base file name */
    WORD Length;            /* Length of data */
    LONG Registration;      /* Registration number */
    BYTE Reserved[8];       /* Unused */
    BYTE Compression;       /* Data compression flag */
} IMAGICHEAD;
```

Id is the identification value for this format and contains the characters IMDC.

Resolution specifies the resolution of the image data. Values are:

00h Low resolution
01h Medium resolution
02h High resolution

Palette is an array of 16 elements storing the color palette for this image.

Date and Time contain a date and time stamp indicating when the file was created. These stamps are in GEMDOS (Atari native operating system) format.

Name is the base filename of the image file.

Length is the length of the image data stored in the file.

Registration is the registration number of the Imagic application program which created the image file.

Reserved is an 8-byte field which is unused and set to a value of 00h.

Compression indicates whether the image data is compressed. A value of 00h indicates no compression, while a value of 01h indicates that the image data is compressed.

Image data may be run-length encoded (RLE) or delta compressed. Delta compression results in smaller animation files than RLE, although on complex images RLE works better.

NEOchrome Format (.NEO)

NEOchrome image files have the file extension .NEO and contain a 79-byte header followed by 16,000 bytes of image data. The format of the header is as follows:

```
typedef struct _NeochromeHeader
{
    WORD Flag;              /* Flag byte (always 00h) */
    WORD Resolution;        /* Image resolution */
    WORD Palette[16];       /* Color palette */
    CHAR FileName[12];      /* Name of image file */
```


Ex.1007 / Page 297 of 1158
APPLE INC.

```
    WORD Limits;            /* Color animation limits */
    WORD Speed;             /* Color animation speed and direction */
    WORD NumberOfSteps;     /* Number of color steps */
    WORD XOffset;           /* Image X offset (always 00h) */
    WORD YOffset;           /* Image Y offset (always 00h) */
    WORD Width;             /* Image width (always 320) */
    WORD Height;            /* Image height (always 200) */
    WORD Reserved[33];      /* Reserved (always 00h) */
} NEOCHROMEHEAD;
```

Flag is a collection of flag bits and is always set to a value of 00h.

Resolution specifies the resolution of the image data. Values are:

00h Low resolution
01h Medium resolution
02h High resolution

Palette is the color palette for this image stored as an array of 16 WORD values.

FileName is the name of the image file. The default string for this field is " . ".

Limits specifies the color animation limits of the image. Bits 0 through 3 specify the value of the upper-right limit, and bits 4 through 7 specify the value of the lower-left limit. Bit 15 is set to 1 if the animation data is valid.

Speed specifies the color animation speed and direction. Bits 0 through 7 specify the speed of the playback in number of blank frames displayed per animation frame. Bit 15 indicates the direction of playback. A value of 0 indicates normal and a value of 1 indicates reversed.

NumberOfSteps is the number of frames in the animation.

XOffset and YOffset are the starting coordinates of the image on the display. These values are always 00h.

Width is the width of the image in pixels. This value is always 320.

Height is the height of the image in pixels. This value is always 200.

Reserved is a field of 33 bytes reserved for future header fields. All bytes in this field are set to 00h.

NEOchrome Animation Format (.ANI)

NEOchrome Animation files have the file extension .ANI and contain a header followed by a sequence of one or more frames of animation data stored in their playback order. The header is 22 bytes and is formatted as follows:

Ex.1007 / Page 298 of 1158
APPLE INC.

```
typedef struct _NewchromeAniHeader
{
    LONG MagicNumber;       /* ID value (always BABEEBEAh) */
    WORD Width;             /* Width of image in bytes */
    WORD Height;            /* Height of image in scan lines */
    WORD Size;              /* Size of image in bytes + 10 */
    WORD XCoord;            /* X coordinate of image */
    WORD YCoord;            /* Y coordinate of image - 1 */
    WORD NumberOfFrames;    /* Total number of frames */
    WORD Speed;             /* Animation playback speed */
    LONG Reserved;          /* Reserved (always 00h) */
} NEWCHROMEANIHEAD;
```

MagicNumber is the identification value for Neochrome animation files. This value is always BABEEBEAh.

Width is the width of the animation in pixels. This value must always be divisible by 8.

Height is the height of the animation frames in pixels (scan lines).

Size is the total size of a frame in bytes, plus 10.

XCoord specifies the left position of the image in pixels, minus one. This value must be divisible by 16.

YCoord specifies the top position of the image in pixels, minus one.

NumberOfFrames specifies the number of image frames in the animation.

Speed specifies the playback speed of the animation in number of blank frames displayed per image frames.

Reserved is an unused field, which is set to 00h.

STAD Format (.PAC)

The STAD image file format has the file extension .PAC. It contains a header followed by a single block of RLE-compressed image data. The header is seven bytes in size and contains only information necessary to decompress the image data. The format of the header is as follows:

```
typedef struct _StadHeader
{
    CHAR Packed[4];         /* Packing orientation of image data */
    BYTE IdByte;            /* RLE ID value of a 'PackByte' run */
    BYTE PackByte;          /* The value of a 'PackByte' run */
    BYTE SpecialByte;       /* RLE ID value of a non-'PackByte' run */
} STADHEADER;
```


Ex.1007 / Page 299 of 1158
APPLE INC.

Packed contains the characters pM86 if the image data in the file is vertically packed, or pM85 if it is horizontally packed.

IdByte is a value used to indicate an RLE byte run that uses the PackByte value.

PackByte is the most frequently occurring byte value in the image data.

SpecialByte is a value used to indicate an RLE byte run using a value stored in the image data.

The image data in a STAD file is always compressed using a simple RLE algorithm. STAD is a bit unique in that it allows the option of packing image data either horizontally along the bitmap (with the scan lines), or vertically down the bitmap (across the scan lines). The direction of the encoding is specified in the Packed field in the header.

The most frequently occurring byte value in the image data is stored in the PackByte field of the header. This reduces the size of the compressed data by not requiring this value to be redundantly stored in the compressed image data itself.

The STAD RLE algorithm uses three types of packets:

- The first type of packet is two bytes in length and contains an Id value and a run-count value. If the ID matches the value stored in the IdByte field of the header, then the value in the PackByte header field is repeated "run count + 1" times. This packet is used only to encode byte runs of the value stored in the PackByte header field.
- The second type of packet is three bytes in length and is used to store a run of a value other than that in the PackByte field. The first byte is an ID matching the SpecialByte field in the header. The second byte is the value of the run, and the third byte is the number of bytes in the run.
- The third type of packet is a single literal byte. If an ID byte is read, and it does not match either the IdByte or SpecialByte value, then this byte is simply written literally to the output.

Following is a simple, pseudo-code description of the RLE decoding process:

```
Read a byte
  If the byte is the IdByte value
    Read a byte (the RunCount)
    Repeat the PackByte value RunCount + 1 times else
```


Ex.1007 / Page 300 of 1158
APPLE INC.

```
If the byte is the SpecialByte value
   Read a byte (the RunValue)
   Read a byte (the RunCount)
   Repeat the RunValue RunCount times
else
   Use the byte value literally.
```

Tiny Format (.TNY, .TN1, .TN2, .TN3)

The Tiny format (.TNY) is similar to the NEOchrome formats. Tiny files may contain low (.TN1), medium (.TN2), or high (.TN3) resolution image data.

Tiny files may have one of two different header formats. The most common is 37 bytes in length and is formatted as follows:

```
typedef struct _TinyHeader
{
    BYTE Resolution;        /* Resolution of the image data */
    WORD Palette[16];       /* Color palette */
    WORD ControlBytes;      /* Number of control bytes */
    WORD DataWords;         /* Number of data words */
} TINYHEAD;
```

Resolution specifies the resolution of the image data. Values are:

00h	Low resolution
01h	Medium resolution
02h	High resolution

Palette is the 16-color palette of the image data.

ControlBytes is the number of control bytes found in the image data. This value is in the range of 3 to 10,667.

DataWords is the number of WORDs of image data stored in the file. This value is in the range of 1 to 16,000.

If the value of the Resolution field is 03h or greater, the Tiny header has the following format:

```
typedef struct _TinyHeader
{
    BYTE Resolution;        /* Resolution of the image data */
    BYTE Limits;            /* Color animation limits */
    BYTE Speed;             /* Speed and direction of playback */
    WORD Duration;          /* Color rotation duration */
    WORD Palette[16];       /* Color palette */
```


Ex.1007 / Page 301 of 1158
APPLE INC.

```
    WORD ControlBytes;      /* Number of control bytes */
    WORD DataWords;         /* Number of data words */
} TINYHEAD;
```

Limits specifies the left and right color animation limits. Bits 0 through 3 store the right (end) limit value, and bits 4 through 7 store the left (start) limit value.

Speed specifies the speed and direction of the animation playback. A negative value indicates left playback, and a positive value indicates right playback. The absolute value is the speed (delay between frames) in increments of 1/60 of a second.

Duration specifies the color rotation duration (number of iterations).

ControlBytes specifies the size of a BYTE array that follows the header. This array contains control values used to specify how the Tiny image data is to be uncompressed.

DataWords specifies the number of data words in the image data. The run-length encoded image data follows the control-value array. Tiny image data is uncompressed by reading a control value and then, based on the control value, performing an action on the encoded data.

If a control value is negative, then the absolute value of the control value indicates the number of WORDs to read literally from the compressed data. If a control value is equal to zero, another control value is read, and its value (128 to 32767) is used to specify the number of times to repeat the next data WORD. If a control value is equal to one, another control value is read and its value (128 to 32767) is used to specify the number of literal WORDs to read from the data section. And a control value greater than one specifies the number of times to repeat the next WORD read from the data section (two to 127).

The uncompressed image data is stored along its Y-axis in a fashion identical to many other Atari image file formats.

For Further Information

Although we have not been able to obtain an official specification document from Atari, the article included on the CD-ROM that accompanies this book contains detailed information about the Atari ST graphics file formats. See the following:

Baggett, David M., *Atari ST Picture Formats.*

Ex.1007 / Page 302 of 1158
APPLE INC.

The author has also kindly agreed to be a resource for information about the Atari ST format files. Contact:

David M. Baggett
Email: *dmb@ai.mit.edu*

You may also find Atari information at:

comp.sys.atari.st
USENET newsgroup

http://newton.ex.ac.uk/general/ug/jones/
Dan's Atari ST web pages

http://www.smartpages.com/faqs/csas-faq/top.html
Atari ST FAQ

Ex.1007 / Page 303 of 1158
APPLE INC.

AutoCAD DXF

NAME:	AutoCAD DXF*
ALSO KNOWN AS:	AutoCAD Drawing Interchange Format, DXF, .DXB, .SLD, .ADI
TYPE:	Vector
COLORS:	256
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Multiple
ORIGINATOR:	Autodesk
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	AutoCAD, many CAD programs, CorelDraw!, others
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Storage and exchange of CAD and vector information.

COMMENTS: A difficult format, mainly because it can contain so many different types of data. The format is controlled and defined by Autodesk for use in its CAD program AutoCAD. The most common form of DXF is 7-bit text, but there are also two related binary formats, one that also uses the DXF extension and another that uses the DXB extension.

Overview

The AutoCAD DXF (Drawing Interchange Format) and the AutoCAD DXB (Drawing Interchange Binary) formats are associated with the CAD application AutoCAD, created and maintained by Autodesk. DXB is a simplified binary version of a DXF file. Other file formats associated with AutoCAD are the slide (.SLD) and plot (.ADI) formats.

* Our thanks to John Foust for his contributions to this article.

Ex.1007 / Page 304 of 1158
APPLE INC.

Although DXF was developed to represent the data used in a CAD program, it is used by many programs as a "least common denominator" format for the exchange of many different types of data, most commonly vector-oriented information but also text and 3D polygons. As a CAD format, it can also express common drafting concepts such as associative dimensions.

Almost any type of data can be represented somehow in DXF. For example, a drawing program like CorelDraw! exports the outlines of the drawing with the AutoCAD POLYLINE entity, while a 3D program might only export 3DFACE entities representing three- and four-sided polygons. DXF also allows a perplexing number of ways of doing nearly the same thing, such as describing objects as separate editable groups. One program might place the objects on different layers of the drawing, while another might use different pen colors, while a third might use named "blocks" to group the data.

Although DXF is widely used for the exchange of simple line data, an application designer wishing to support DXF must consider that AutoCAD can store these many types of data in different ways.

Sometimes the correct interpretation of a DXF file can be very difficult. The intended appearance of lines and regions can be dependent on many seemingly obscure settings in the header of the DXF file. Because DXF files are so difficult to faithfully interpret, many application designers decide to only export DXF.

Even among programs that claim to import DXF, you may find they only support a subset of everything that is possible in DXF. If you hope to create your own DXF files in order to transfer data into a program that claims to import DXF, be sure you know which representations it understands.

With each new version of AutoCAD, DXF changes. AutoCAD Release 13 expanded the DXF format in many ways in order to represent the specialized data of a new geometry engine. These additions store the complex surface and solid information for Spatial Technology's ACIS geometry engine, now a part of AutoCAD. Not all of this information has been documented and must be skipped by any DXF reader. With Release 13, AutoCAD's own tolerance for minimal DXF files changed, too, as it expanded an auditing step that checks the validity of the DXF files it imports.

Obviously, the DXF file format is quite complex and subtle. Because it would take more than 50 pages to fully document every possible part of this format, we will simply outline the basic structure of any DXF file. For full details, refer to Autodesk's specification document on the CD-ROM included with this book.

Ex.1007 / Page 305 of 1158
APPLE INC.

File Organization

A DXF file consists of up to seven sections: a header, tables, blocks, classes, objects, entities, and an end-of-file marker.

- The HEADER section contains variables that represent the state of AutoCAD's internal settings. For example, the AutoCAD version variable "$ACADVER" is set to "AC1012" in a DXF file saved by AutoCAD Release 13. Other variables set the units used to measure angles, defaults for chamfering, offsets, and scalings, etc.
- The TABLES section contains several lists of information used in the rest of the drawing, such as the list of line types, layer names, fonts, and preset views of the drawing.
- The BLOCKS section contains predefined drawing elements that might be present in the drawing. For example, a block could define a standard door knob that is placed on every door in a drawing. Block definitions are referenced in the ENTITIES section with the INSERT command.
- The CLASSES and OBJECTS sections were introduced with AutoCAD Release 13. The CLASSES section holds the description of any application-defined classes of objects that may be instantiated in the BLOCKS or ENTITIES sections.
- The OBJECT section contains non-graphical parts of the drawing. All entities that are not part of the entities or symbol tables are "objects." For example, AutoCAD dictionaries are stored here.
- The ENTITIES section contains the actual object data of the drawing. This can include raw data such as LINE and ARC entities as well as INSERT commands that place a predefined block definition at a certain position in the drawing.
- The end of the DXF data is marked with an EOF directive on the last line of the file.

File Details

A DXF file is composed of pairs of group codes and associated values. Each occurs on its own line in the text file. The integer group code indicates the type of the value to follow. Group codes occur in ranges. For example, group codes 0 to 9 are followed by strings, and each different group code is used in certain situations. Group code 0 indicates the start of an entity, table or end-of-file indicator. Code 1 indicates the primary text value for an entity. Group code

Ex.1007 / Page 306 of 1158
APPLE INC.

2 is used for names, such as names of sections, blocks, table names, etc. Code 9 introduces the name of a header section variable. For example, at the start of every DXF file, group code 0 precedes the SECTION command, followed by group code 2 with a string indicating the type of section, such as HEADER:

```
  0
SECTION
  2
HEADER
  9
$ACADVER
  1
AC1012
```

Ranges of group codes indicate the type of data to follow. Group codes 10 to 59 are used for floating-point values, such as point coordinates. Codes 60 to 79 store integer values. For example, to store a 2D point location, first group code 10 is used for the X value, then code 20 is used for the Y value. If the entity has a secondary coordinate value, it would also use group codes 11 and 21. Here is a minimal yet complete DXF file that describes a line from location (1,2) to (3,4):

```
0
SECTION
  2
ENTITIES
999
This is just a line.
  0
LINE
  8
0
 10
1.0
20
2.0
11
3.0
21
4.0
  0
ENDSEC
  0
EOF
```

Group code 999 precedes a comment. This line will be placed on layer 0, as indicated by group code 8. This minimal file is an example of an "entities only" file that will be accepted by almost any program that claims to import DXF.

Ex.1007 / Page 307 of 1158
APPLE INC.

As AutoCAD is expanded with each new version, new group codes are added. If you are writing a program that reads DXF files, you can ensure future compatibility by ignoring undefined group code and value pairs.

One curious aspect of DXF is that it does not contain a color palette, yet most objects in a DXF file can be assigned a distinct color value with group code 62. Each drawing entity can be assigned a number from 1 to 255 known as an AutoCAD Color Index, or ACI, also described in earlier documentation as a "pen number." This reflects AutoCAD's origins as a CAD package where drawings were typically printed by a pen plotter that had several ink pens but with no standard correspondence to actual RGB values, or even to the colors of the lines on the screen. AutoCAD now sets a default RGB color for each ACI when it appears onscreen, but these are not stored in the DXF file.

Binary DXF

The most commonly used form of DXF is stored in 7-bit ASCII characters, but a binary format also uses the extension DXF. AutoCAD Release 10 was the first to support binary DXF. Binary DXF files are usually 20 to 30 percent smaller than the ASCII version, and they load more quickly into AutoCAD.

Binary DXF files always begin with a specific 22-byte identification string:

```
AutoCAD Binary DXF<0Dh><0Ah><1Ah><00h>
```

Binary DXF uses group-value pairs, too. Group codes are usually one byte, followed by either a two-byte little-endian integer, an eight-byte IEEE floating-point double value, or a zero-terminated string, depending on the type of value associated with the group code's range. To represent group codes greater than 254, the value 255 precedes a two-byte integer group code.

A third form of DXF known as DXB is an even simpler binary format. DXB files are even smaller than the binary DXF format. DXB files are limited to a small set of entities such as line, point, circle, arc, trace, solid, polyline, and 3D face. Entities are indicated by their own byte code and are immediately followed by the necessary data for that entity, in an appropriate integer or floating-point format.

A DXB file can be distinguished from a binary DXF file by the file extension .DXB and by the fact that it always begins with a 19-byte identification string:

```
AutoCAD DXB 1.0<0Dh><0Ah><1Ah><00h>
```


Ex.1007 / Page 308 of 1158
APPLE INC.

For Further Information

For further information about the AutoCAD DXF format, see the DXF specification included on the CD-ROM that accompanies this book.

The AutoCAD Manual Release 12 also contains complete information on the DXF format; see:

AutoCAD Customization Manual, Release 12, Autodesk Inc., 1992, pp. 241–281.

AutoCAD Developer's Guide, Release 13, Autodesk Inc., 1995, pp. 505–556.

Autodesk has also released an electronic document describing the DXF format, which may be found on many online services and BBSs.

Many books on AutoCAD have been published, and several include in-depth information on the DXF format, including the following:

Jones, Frederick H. and Lloyd Martin, *The AutoCAD Database Book*, Ventana Press, Chapel Hill, NC, 1989.

Johnson, N., *AutoCAD, The Complete Reference*, second edition, McGraw-Hill, New York, NY, 1991.

For additional information about this format, you may also contact:

Autodesk, Inc.
Attn: Neele Johnston
Autodesk Developer Marketing
2320 Marinship Way
Sausalito, CA 94965
Voice: 415-491-8719
Email: *neele@autodesk.com*
WWW: *http://www.autodesk.com/*

Ex.1007 / Page 309 of 1158
APPLE INC.

Autodesk 3D Studio

NAME:	Autodesk 3D Studio
ALSO KNOWN AS:	3DS, ASC
TYPE:	Scene
COLORS:	Unlimited
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	None
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	Multiple
ORIGINATOR:	Autodesk
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	3D Studio, Caligari trueSpace, other 3D programs
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Storage of 3D scene information for use by rendering applications.

COMMENTS: Autodesk 3D Studio has come to be used as an interchange format. Many 3D modeling and rendering applications read and write 3DS, as well as their own formats. Why this is so is not clear but may be due to Autodesk 3D Studio's longevity and visibility in the market. The program implements a large number of functions, however, and this is reflected in the file.

Overview

Autodesk 3D Studio (3DS) is a relatively high-end program used for the modeling and rendering of 3D scenes. It runs under MS-DOS and uses a third-party memory manager to make optimum use of the resources available on the PC platform. Output compares favorably to many higher end platforms. It has thus become one of the platforms of choice for people doing detailed construction of 3D scenes.

Unfortunately, Autodesk has chosen to make information about this format available only through purchase of their developer's kit. Information for this article was accordingly obtained from public sources and is necessarily

Ex.1007 / Page 310 of 1158
APPLE INC.

incomplete. We would appreciate any further information available on this format.

There are two formats used by 3D Studio:

- A binary one designated by the suffix 3DS
- An ASCII format using the suffix ASC

Both are used as interchange formats. This article concentrates on the binary version of the file, because the ASCII version is usually used for the storage of object definitions and closely follows the tags defined in the binary version. At the end of this article, we provide an example of a simple ASC file.

File Organization

3DS files consist of tags, called *chunks* by Autodesk. Each chunk consists of an ID value and an offset to the start of the next chunk. Data associated with each chunk follows the ID and offset information.

```
typedef struct _CHUNK
{
   WORD  Chunk_ID;        /* Tag type */
   DWORD Chunk_length;    /* Relative offset in bytes to next chunk */
} CHUNK;
```

Chunks are classified as either Primary, Main, or Subordinate and are arranged in a hierarchy as follows:

```
Primary Chunk
  Main Chunk #1
    Subordinate Chunk #1
    Subordinate Chunk #2
    :
  Main Chunk #2
    Subordinate Chunk #1
    Subordinate Chunk #2
    :
  Main Chunk #3
  :
```

The Main chunks are said to be "owned" by the Primary chunk, and the Subordinate chunks associated with the Main chunks are owned by the Main chunk. Note that many of the chunks defined below are associated with the behavior

Ex.1007 / Page 311 of 1158
APPLE INC.

of 3D Studio at run-time. They are listed here because your rendering application or file conversion program may be able to make use of them as clues to the appearance and behavior of the objects found in the file.

There is only one Primary chunk per file, located at the start of the file. Following the Primary chunk tag is the first Main tag. The chunk_length field of Main chunk tags is generally an offset to the next Main chunk.

Following each Main chunk tag is data associated with the Main chunk and/or Subordinate chunk tags. Thus, the reader/parser must understand the format and length of the data associated with each Main chunk in order to extract information contained in Subordinate chunks owned by the Main chunk.

File Details

The first tag of the file, the Primary chunk, has an ID of 4D4Dh. Beneath the Primary chunk in the tag hierarchy are Main chunks.

Chunks

Table Autodesk 3D Studio-1 lists the possible chunks known at this time:

TABLE AUTODESK 3D STUDIO-1: ***Autodesk 3D Studio Chunks***

ID	Name	Data Type	Description
0000	NULL_CHUNK	NA	
0001	unknown	(possibly FLOAT)	
0002	M3D_VERSION	SHORT	Version
0005	M3D_KFVERSION	NA	
0010	COLOR_F	FLOAT R,G,B	Floating-point color description
0011	COLOR_24	BYTE R,G,B	24-bit color description
0012	LIN_COLOR_24	BYTE R,G,B	24-bit color description
0013	LIN_COLOR_F	FLOAT R,G,B	Floating-point color description
0030	INT_PERCENTAGE	SHORT	Percentage value
0031	FLOAT_PERCENTAGE	FLOAT	Percentage value
0100	MASTER_SCALE	FLOAT	Scale factor

Ex.1007 / Page 312 of 1158
APPLE INC.

ID	Name	Data Type	Description
0995	ChunkType	NA	
0996	ChunkUnique	NA	
0997	NotChunk	NA	
0998	Container	NA	
0999	IsChunk	NA	
0c3c	C_SXP_SELFI_MASKDATA	NA	
1100	BIT_MAP	CHAR[]	0-terminated ASCII string
1101	USE_BIT_MAP	NA	
1200	SOLID_BGND	FLOAT R,G,B	
1201	USE_SOLID_BGND	NA	
1300	V_GRADIENT	FLOAT	Midpoint of gradient
1301	USE_V_GRADIENT	NA	
1400	LO_SHADOW_BIAS	FLOAT	Bias
1410	HI_SHADOW_BIAS	NA	
1420	SHADOW_MAP_SIZE	SHORT	Size
1430	SHADOW_SAMPLES	NA	
1430	SHADOW_RANGE	NA	
1430	SHADOW_FILTER	FLOAT	Filter
1460	RAY_BIAS	FLOAT	Bias
1500	O_CONSTS	FLOAT x,y,z	Plane values
2100	AMBIENT_LIGHT	NA	
2200	FOG	FLOAT[4]	near_plane, near_density, far_plane, far_density
2201	USE_FOG	NA	
2210	FOG_BGND	NA	
2300	DISTANCE_CUE	FLOAT[4]	near_plane, near_density, far_plane, far_density
2301	USE_DISTANCE_CUE	NA	
2302	LAYER_FOG	FLOAT[3], SHORT	fog_z_from, fog_z_to, fog_density fog_type
2303	USE_LAYER_FOG	NA	

Ex.1007 / Page 313 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
2310	DCUE_BGND	NA	
2d2d	SMAGIC	NA	
2d3d	LMAGIC	NA	
3000	DEFAULT_VIEW	NA	
3010	VIEW_TOP	FLOAT[4]	target_x, target_y, target_z, view_width
3020	VIEW_BOTTOM	FLOAT[4]	target_x, target_y, target_z, view_width
3030	VIEW_LEFT	FLOAT[4]	target_x, target_y, target_z, view_width
3040	VIEW_RIGHT	FLOAT[4]	target_x, target_y, target_z, view_width
3050	VIEW_FRONT	FLOAT[4]	target_x, target_y, target_z, view_width
3060	VIEW_BACK	FLOAT[4]	target_x, target_y, target_z, view_width
3070	VIEW_USER	FLOAT[4]	target_x, target_y, target_z, view_width
3080	VIEW_CAMERA	CHAR[]	0-terminated ASCII string
3090	VIEW_WINDOW	NA	
3d3d	MDATA	WORD	Mesh data magic number 4d4d
3d3e	MESH_VERSION	NA	
3daa	MLIBMAGIC	WORD	Material library magic number
3dc2	PRJMAGIC	WORD	Project number magic number
3dff	MATMAGIC	WORD	Material file magic number
4000	NAMED_OBJECT	CHAR[]	0-terminated ASCII string
4010	OBJ_HIDDEN	NA	
4011	OBJ_VIS_LOFTER	NA	
4012	OBJ_DOESNT_CAST	NA	
4013	OBJ_MATTE	NA	
4014	OBJ_FAST	NA	
4015	OBJ_PROCEDURAL	NA	
4016	OBJ_FROZEN	NA	
4017	OBJ_DONT_RCVSHADOW	NA	

Ex.1007 / Page 314 of 1158
APPLE INC.

ID	Name	Data Type	Description
4100	N_TRI_OBJECT	NA	Named triangle object
4110	POINT_ARRAY	SHORT, POINT[]	Number of points Array of POINT structs
4111	POINT_FLAG_ARRAY	SHORT, SHORT[]	Number of flags Array of SHORT flags
4120	FACE_ARRAY	SHORT, FACE[]	Number of faces Array of FACE structs
4130	MSH_MAT_GROUP	CHAR[], SHORT, SHORT[]	Material name Number of faces Face numbers
4140	TEX_VERTS	SHORT, VERTEX[]	Number of vertices Array of VERTEX structs
4150	SMOOTH_GROUP	SHORT[]	Groups = faces x 4
4160	MESH_MATRIX	FLOAT[4][3]	4x3 FLOAT matrix
4170	MESH_TEXTURE_INFO	SHORT, FLOAT[2], FLOAT[3], FLOAT[4][3], FLOAT[4]	map_type x_tiling, y_tiling icon_x, icon_y, icon_z 4x3 FLOAT matrix scaling, plan_icon_w, plan_icon_h, cyl_icon_h
4181	PROC_NAME	NA	
4182	PROC_DATA	NA	
4190	MSH_BOXMAP	NA	
4400	N_D_L_OLD	NA	
4500	N_CAM_OLD	NA	
4600	N_DIRECT_LIGHT	FLOAT[3]	x,y,z of light
4610	DL_SPOTLIGHT	FLOAT[5]	target_x, target_y, target_z, hotspot_ang, falloff_ang
4620	DL_OFF	NA	
4625	DL_ATTENUATE	NA	

Ex.1007 / Page 315 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
4627	DL_RAYSHAD	NA	
4630	DL_SHADOWED	NA	
4640	DL_LOCAL_SHADOW	NA	
4641	DL_LOCAL_SHADOW2	NA	
4650	DL_SEE_CONE	NA	
4651	DL_SPOT_RECTANGULAR	NA	
4652	DL_SPOT_OVERSHOOT	NA	
4653	DL_SPOT_PROJECTOR	NA	
4654	DL_EXCLUDE	NA	
4655	DL_RANGE	NA	
4656	DL_SPOT_ROLL	FLOAT	Roll angle
4657	DL_SPOT_ASPECT	NA	
4658	DL_RAY_BIAS	FLOAT	Bias
4659	DL_INNER_RANGE	FLOAT	Range
465a	DL_OUTER_RANGE	FLOAT	Range
465b	DL_MULTIPLIER	FLOAT	Multiplier value
4680	N_AMBIENT_LIGHT	NA	
4700	N_CAMERA	FLOAT[3],	camera_x, camera_y, camera_z
		FLOAT[3],	target_x, target_y, target_z
		FLOAT[2]	bank_angle, focus
4710	CAM_SEE_CONE	NA	
4720	CAM_RANGES	FLOAT[2]	near_range, far_range
4d4d	M3DMAGIC	NA	3DS file magic number
4f00	HIERARCHY	NA	
4f10	PARENT_OBJECT	NA	
4f20	PIVOT_OBJECT	NA	
4f30	PIVOT_LIMITS	NA	
4f40	PIVOT_ORDER	NA	
4f50	XLATE_RANGE	NA	
5000	POLY_2D	NA	
5010	SHAPE_OK	NA	
5011	SHAPE_NOT_OK	NA	
5020	SHAPE_HOOK	NA	
6000	PATH_3D	NA	
6005	PATH_MATRIX	NA	
6010	SHAPE_2D	NA	

Ex.1007 / Page 316 of 1158
APPLE INC.

ID	Name	Data Type	Description
6020	M_SCALE	NA	
6030	M_TWIST	NA	
6040	M_TEETER	NA	
6050	M_FIT	NA	
6060	M_BEVEL	NA	
6070	XZ_CURVE	NA	
6080	YZ_CURVE	NA	
6090	INTERPCT	NA	
60a0	DEFORM_LIMIT	NA	
6100	USE_CONTOUR	NA	
6110	USE_TWEEN	NA	
6120	USE_SCALE	NA	
6130	USE_TWIST	NA	
6140	USE_TEETER	NA	
6150	USE_FIT	NA	
6160	USE_BEVEL	NA	
7000	VIEWPORT_LAYOUT_OLD	NA	
7001	VIEWPORT_LAYOUT	SHORT[7]	form, top, ready, wstate, swapws, swapport, swapcur
7010	VIEWPORT_DATA_OLD	NA	
7011	VIEWPORT_DATA	SHORT[7],	flags, axis_lockout, win_x, win_y, win_w, win_h, win_view
		FLOAT[6],	zoom, worldcenter_x, worldcenter_y, worldcenter_z, horiz_and, vert_ang
		CHAR[]	0-terminated ASCII string—camera name
7011	VIEWPORT_DATA_3	SHORT[7],	flags, axis_lockout, win_x, win_y, win_w, win_h, win_view
		FLOAT[6],	zoom, worldcenter_x, worldcenter_y, worldcenter_z, horiz_and, vert_ang

Ex.1007 / Page 317 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
		CHAR[]	0-terminated ASCII string—camera name
7020	VIEWPORT_SIZE	SHORT[4]	x,y,width,height
7030	NETWORK_VIEW	NA	
8000	XDATA_SECTION	NA	
8001	XDATA_ENTRY	NA	
8002	XDATA_APPNAME	NA	
8003	XDATA_STRING	NA	
8004	XDATA_FLOAT	NA	
8005	XDATA_DOUBLE	NA	
8006	XDATA_SHORT	NA	
8007	XDATA_LONG	NA	
8008	XDATA_VOID	NA	
8009	XDATA_GROUP	NA	
800a	XDATA_RFU6	NA	
800b	XDATA_RFU5	NA	
800c	XDATA_RFU4	NA	
800d	XDATA_RFU3	NA	
800e	XDATA_RFU2	NA	
800f	XDATA_RFU1	NA	
80f0	PARENT_NAME	NA	
a000	MAT_NAME	CHAR[]	0-terminated ASCII string—material name
a010	MAT_AMBIENT	NA	
a020	MAT_DIFFUSE	NA	
a030	MAT_SPECULAR	NA	
a040	MAT_SHININESS	NA	
a041	MAT_SHIN2PCT	NA	
a042	MAT_SHIN3PCT	NA	
a050	MAT_TRANSPARENCY	NA	
a052	MAT_XPFALL	NA	
a053	MAT_REFBLUR	NA	
a080	MAT_SELF_ILLUM	NA	
a081	MAT_TWO_SIDE	NA	
a082	MAT_DECAL	NA	
a083	MAT_ADDITIVE	NA	
a084	MAT_SELF_ILPCT	NA	
a085	MAT_WIRE	NA	

Ex.1007 / Page 318 of 1158
APPLE INC.

ID	Name	Data Type	Description
a086	MAT_SUPERSMP	NA	
a087	MAT_WIRESIZE	FLOAT	Wire size
a088	MAT_FACEMAP	NA	
a08a	MAT_XPFALLIN	NA	
a08c	MAT_PHONGSOFT	NA	
a08e	MAT_WIREABS	NA	
a100	MAT_SHADING	SHORT	Shading value
a200	MAT_TEXMAP	NA	
a204	MAT_SPECMAP	NA	
a210	MAT_OPACMAP	NA	
a220	MAT_REFLMAP	NA	
a230	MAT_BUMPMAP	NA	
a240	MAT_USE_XPFALL	NA	
a250	MAT_USE_REFBLUR	NA	
a252	MAT_BUMP_PERCENT	NA	
a300	MAT_MAPNAME	CHAR[]	0-terminated ASCII filename
a310	MAT_ACUBIC	NA	
a320	MAT_SXP_TEXT_DATA	NA	
a321	MAT_SXP_TEXT2_DATA	NA	
a322	MAT_SXP_OPAC_DATA	NA	
a324	MAT_SXP_BUMP_DATA	NA	
a325	MAT_SXP_SPEC_DATA	NA	
a326	MAT_SXP_SHIN_DATA	NA	
a328	MAT_SXP_SELF1_DATA	NA	
a32a	MAT_SXP_TEXT_DATA	NA	
a32c	MAT_SXP_TEXT2_MASKDATA	NA	
a32e	MAT_SXP_OPAC_MASKDATA	NA	
a330	MAT_SXP_BUMP_MASKDATA	NA	
a332	MAT_SXP_SPEC_MASKDATA	NA	
a334	MAT_SXP_SHIN_MASKDATA	NA	
a336	MAT_SXP_SELF1_MASKDATA	NA	
a338	MAT_SXP_REFL_MASKDATA	NA	
a33a	MAT_TEX2MAP	NA	
a33c	MAT_SHINMAP	NA	
a33d	MAT_SELFIMAP	NA	
a33e	MAT_TEXMASK	NA	
a340	MAT_TEX2MASK	NA	
a342	MAT_OPACMASK	NA	
a344	MAT_BUMPMASK	NA	

Ex.1007 / Page 319 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
a346	MAT_SHINMASK	NA	
a348	MAT_SPECMASK	NA	
a34a	MAT_SELFIMASK	NA	
a34c	MAT_REFLMASK	NA	
a350	MAT_MAP_TILINGOLD	NA	
a351	MAT_MAP_TILING	SHORT	Flags
a352	MAT_MAP_TEXBLUR_OLD	NA	
a353	MAT_MAP_TEXBLUR	FLOAT	Blurring value
a354	MAT_MAP_USCALE	NA	
a356	MAT_MAP_VSCALE	NA	
a358	MAT_MAP_UOFFSET	NA	
a35a	MAT_MAP_VOFFSET	NA	
a35c	MAT_MAP_ANG	NA	
a360	MAT_MAP_COL1	NA	
a362	MAT_MAP_COL2	NA	
a364	MAT_MAP_RCOL	NA	
a366	MAT_MAP_GCOL	NA	
a368	MAT_MAP_BCOL	NA	
afff	MAP_ENTRY	NA	
b000	KFDATA	NA	
b001	AMBIENT_NODE_TAG	NA	
b002	OBJECT_NODE_TAG	NA	
b003	CAMERA_NODE_TAG	NA	
b004	TARGET_NODE_TAG	NA	
b005	LIGHT_NODE_TAG	NA	
b006	L_TARGET_NODE_TAG	NA	
b007	SPOTLIGHT_NODE_TAG	NA	
b008	KFSEG	SHORT[2]	start, end
b009	KFCURTIME	SHORT	curframe
b00a	KFHDR	SHORT,	Revision number
		CHAR[],	0-terminated ASCII filename
		SHORT	Animation length
b010	NODE_HDR	CHAR[],	0-terminated ASCII object name
		SHORT[3]	flags1, flags2, hierarchy
b011	INSTANCE_NAME	NA	
b012	PRESCALE	NA	

Ex.1007 / Page 320 of 1158
APPLE INC.

ID	Name	Data Type	Description
b013	PIVOT	FLOAT[3]	pivot_x, pivot_y, pivot_z
b014	BOUNDBOX	NA	
b015	MORPH_SMOOTH	FLOAT	Smoothing angle (radians)
b020	POS_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT	Unknown
		POSITION[keys]	Array of POSITION structs
b021	ROT_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		ROTATION[keys]	Array of ROTATION structs
b022	SCL_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		SCALE[keys]	Array of SCALE structs
b023	FOV_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		FOV[keys]	Array of FOV structs
b024	ROLL_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		ROLL[keys]	Array of ROLL structs
b025	COL_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		COLOR[keys]	Array of COLOR structs

Ex.1007 / Page 321 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
b026	MORPH_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		MORPH[keys]	Array of MORPH structs
b027	HOT_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		HOTSPOT[keys]	Array of HOTSPOT structs
b028	FALL_TRACK_TAG	SHORT,	Flags
		SHORT[4],	Unknown
		SHORT,	Keys
		SHORT,	Unknown
		FALLOFF[keys]	Array of FALLOFF structs
b029	HIDE_TRACK_TAG	NA	
b030	NODE_ID	SHORT	ID value
c010	C_MDRAWER	NA	
c020	C_TDRAWER	NA	
c030	C_SHPDRAWER	NA	
c040	C_MODDRAWER	NA	
c050	C_RIPDRAWER	NA	
c060	C_TXDDRAWER	NA	
c062	C_PDRAWER	NA	
c064	C_MTLDRAWER	NA	
c066	C_FLIDRAWER	NA	
c067	C_CUBDRAWER	NA	
c070	C_MFILE	NA	
c080	C_SHPFILE	NA	
c090	C_MODFILE	NA	
c0a0	C_RIPFILE	NA	
c0b0	C_TXFILE	NA	
c0b2	C_PFILE	NA	
c0b4	C_MTLFILE	NA	
c0b6	C_FLIFILE	NA	
c0b8	C_PALFILE	NA	
c0c0	C_TX_STRING	NA	

Ex.1007 / Page 322 of 1158
APPLE INC.

ID	Name	Data Type	Description
c0d0	C_CONSTS	NA	
c0e0	C_SNAPS	NA	
c0f0	C_GRIDS	NA	
c100	C_ASNAPS	NA	
c110	C_GRID_RANGE	NA	
c120	C_RENDTYPE	NA	
c130	C_PROGMODE	NA	
c140	C_PREVMODE	NA	
c150	C_MODWMODE	NA	
c160	C_MODMODEL	NA	
c170	C_ALL_LINES	NA	
c180	C_BACK_TYPE	NA	
c190	C_MD_CS	NA	
c1a0	C_MD_CE	NA	
c1b0	C_MD_SML	NA	
c1c0	C_MD_SMW	NA	
c1c3	C_LOFT_WITH_TEXTURE	NA	
c1c4	C_LOFT_L_REPEAT	NA	
c1c5	C_LOFT_W_REPEAT	NA	
c1c6	C_LOFT_UV_NORMALIZE	NA	
c1c7	C_WELD_LOFT	NA	
c1d0	C_MD_PDET	NA	
c1e0	C_MD_SDET	NA	
c1f0	C_RGB_MODE	NA	
c200	C_RGB_HIDE	NA	
c202	C_RGB_MAPSW	NA	
c204	C_RGB_TWOSIDE	NA	
c208	C_RGB_SHADOW	NA	
c210	C_RGB_AA	NA	
c220	C_RGB_OVW	NA	
c230	C_RGB_OVH	NA	
c23d	CMAGIC	NA	
c240	C_RGB_PICTYPE	NA	
c250	C_RGB_OUTPUT	NA	
c253	C_RGB_TODISK	NA	
c254	C_RGB_COMPRESS	NA	
c255	C_JPEG_COMPRESSION	NA	
c256	C_RGB_DISPDEV	NA	
c259	C_RGB_HARDDEV	NA	
c25a	C_RGB_PATH	NA	

Ex.1007 / Page 323 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
c25b	C_BITMAP_DRAWER	NA	
c260	C_RGB_FILE	NA	
c270	C_RGB_OVASPECT	NA	
c271	C_RGB_ANIMTYPE	NA	
c272	C_RENDER_ALL	NA	
c273	C_REND_FROM	NA	
c274	C_REND_TO	NA	
c275	C_REND_NTH	NA	
c276	C_PAL_TYPE	NA	
c277	C_RND_TURBO	NA	
c278	C_RND_MIP	NA	
c279	C_BGND_METHOD	NA	
c27a	C_AUTO_REFLECT	NA	
c27b	C_VP_FROM	NA	
c27c	C_VP_TO	NA	
c27d	C_VP_NTH	NA	
c27e	C_REND_TSTEP	NA	
c27f	C_VP_TSTEP	NA	
c280	C_SRDIAM	NA	
c290	C_SRDEG	NA	
c2a0	C_SRSEG	NA	
c2b0	C_SRDIR	NA	
c2c0	C_HETOP	NA	
c2d0	C_HEBOT	NA	
c2e0	C_HEHT	NA	
c2f0	C_HETURNS	NA	
c300	C_HEDEG	NA	
c310	C_HESEG	NA	
c320	C_HEDIR	NA	
c330	C_QUIKSTUFF	NA	
c340	C_SEE_LIGHTS	NA	
c350	C_SEE_CAMERAS	NA	
c360	C_SEE_3D	NA	
c370	C_MESHSEL	NA	
c380	C_MESHUNSEL	NA	
c390	C_POLYSEL	NA	
c3a0	C_POLYUNSEL	NA	
c3a2	C_SHPLOCAL	NA	
c3a4	C_MSHLOCAL	NA	
c3b0	C_NUM_FORMAT	NA	

Ex.1007 / Page 324 of 1158
APPLE INC.

ID	Name	Data Type	Description
c3c0	C_ARCH_DENOM	NA	
c3d0	C_IN_DEVICE	NA	
c3e0	C_MSCALE	NA	
c3f0	C_COMM_PORT	NA	
c400	C_TAB_BASES	NA	
c410	C_TAB_DIVS	NA	
c420	C_MASTER_SCALES	NA	
c430	C_SHOW_1STVERT	NA	
c440	C_SHAPER_OK	NA	
c450	C_LOFTER_OK	NA	
c460	C_EDITOR_OK	NA	
c470	C_KEYFRAMER_OK	NA	
c480	C_PICKSIZE	NA	
c490	C_MAPTYPE	NA	
c4a0	C_MAP_DISPLAY	NA	
c4b0	C_TILE_XY	NA	
c4c0	C_MAP_XYZ	NA	
c4d0	C_MAP_SCALE	NA	
c4e0	C_MAP_MATRIX_OLD	NA	
c4e1	C_MAP_MATRIX	NA	
c4f0	C_MAP_WID_HT	NA	
c500	C_OBNAME	NA	
c510	C_CAMNAME	NA	
c520	C_LTNAME	NA	
c525	C_CUR_MNAME	NA	
c526	C_CURMTL_FROM_MESH	NA	
c527	C_GET_SHAPE_MAKE_FACES	NA	
c530	C_DETAIL	NA	
c540	C_VERTMARK	NA	
c550	C_MSHAX	NA	
c560	C_MSHCP	NA	
c570	C_USERAX	NA	
c580	C_SHOOK	NA	
c590	C_RAX	NA	
c5a0	C_STAPE	NA	
c5b0	C_LTAPE	NA	
c5c0	C_ETAPE	NA	
c5c8	C_KTAPE	NA	
c5c0	C_ETAPE	NA	
c5d0	C_SPHSEGS	NA	

Ex.1007 / Page 325 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
c5e0	C_GEOSMOOTH	NA	
c5f0	C_HEMISEGS	NA	
c600	C_PRISMSEGS	NA	
c610	C_PRISMSIDES	NA	
c620	C_TUBESEGS	NA	
c630	C_TUBESIDES	NA	
c640	C_TORSEGS	NA	
c650	C_TORSIDES	NA	
c660	C_CONESIDES	NA	
c661	C_CONESEGS	NA	
c670	C_NGPARMS	NA	
c680	C_PTHLEVEL	NA	
c690	C_MSCSYM	NA	
c6a0	C_MFTSYM	NA	
c6b0	C_MTTSYM	NA	
c6c0	C_SMOOTHING	NA	
c6d0	C_MODICOUNT	NA	
c6e0	C_FONTSEL	NA	
c6f0	C_TESS_TYPE	NA	
c6f1	C_TESS_TENSION	NA	
c700	C_SEG_START	NA	
c705	C_SEG_END	NA	
c710	C_CURTIME	NA	
c715	C_ANIMLENGTH	NA	
c720	C_PV_FROM	NA	
c725	C_PV_TO	NA	
c730	C_PV_DOFNUM	NA	
c735	C_PV_RNG	NA	
c740	C_PV_NTH	NA	
c745	C_PV_TYPE	NA	
c750	C_PV_METHOD	NA	
c755	C_PV_FPS	NA	
c765	C_VTR_FRAMES	NA	
c770	C_VTR_HDTL	NA	
c771	C_VTR_HD	NA	
c772	C_VTR_TL	NA	
c775	C_VTR_IN	NA	
c780	C_VTR_PK	NA	
c785	C_VTR_SH	NA	
c790	C_WORK_MTLS	NA	

Ex.1007 / Page 326 of 1158
APPLE INC.

ID	Name	Data Type	Description
c792	C_WORK_MTLS_2	NA	
c793	C_WORK_MTLS_3	NA	
c794	C_WORK_MTLS_4	NA	
c7a1	C_BGTYPE	NA	
c7b0	C_MEDTILE	NA	
c7d0	C_LO_CONTRAST	NA	
c7d1	C_HI_CONTRAST	NA	
c7e0	C_FROZ_DISPLAY	NA	
c7f0	C_BOOLWELD	NA	
c7f1	C_BOOLTYPE	NA	
c900	C_AND_THRESH	NA	
c901	C_SS_THRESH	NA	
c903	C_TEXTURE_BLUR_DEFAULT	NA	
ca00	C_MAPDRAWER	NA	
ca01	C_MAPDRAWER1	NA	
ca02	C_MAPDRAWER2	NA	
ca03	C_MAPDRAWER3	NA	
ca04	C_MAPDRAWER4	NA	
ca05	C_MAPDRAWER5	NA	
ca06	C_MAPDRAWER6	NA	
ca07	C_MAPDRAWER7	NA	
ca08	C_MAPDRAWER8	NA	
ca09	C_MAPDRAWER9	NA	
ca10	C_MAPDRAWER_ENTRY	NA	
ca20	C_BACKUP_FILE	NA	
ca21	C_DITHER_256	NA	
ca22	C_SAVE_LAST	NA	
ca23	C_USE_ALPHA	NA	
ca24	C_TGA_DEPTH	NA	
ca25	C_REND_FIELDS	NA	
ca26	C_REFLIP	NA	
ca27	C_SEL_ITEMTOG	NA	
ca28	C_SEL_RESET	NA	
ca29	C_STICKY_KEYINF	NA	
ca2a	C_WELD_THRESHOLD	NA	
ca2b	C_ZCLIP_POINT	NA	
ca2c	C_ALPHA_SPLIT	NA	
ca30	C_KF_SHOW_BACKFACE	NA	
ca40	C_OPTIMIZE_LOFT	NA	
ca42	C_TENS_DEFAULT	NA	

Ex.1007 / Page 327 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
ca44	C_CONT_DEFAULT	NA	
ca46	C_BIAS_DEFAULT	NA	
ca50	C_DXFNAME_SRC	NA	
ca60	C_AUTO_WELD	NA	
ca70	C_AUTO_UNIFY	NA	
ca80	C_AUTO_SMOOTH	NA	
ca90	C_DXF_SMOOTH_ANG	NA	
caa0	C_SMOOTH_ANG	NA	
cb00	C_WORK_MTLS_5	NA	
cb01	C_WORK_MTLS_6	NA	
cb02	C_WORK_MTLS_7	NA	
cb03	C_WORK_MTLS_8	NA	
cb04	C_WORKMTL	NA	
cb10	C_SXP_TEXT_DATA	NA	
cb11	C_SXP_OPAC_DATA	NA	
cb12	C_SXP_BUMP_DATA	NA	
cb13	C_SXP_SHIN_DATA	NA	
cb20	C_SXP_TEXT2_DATA	NA	
cb24	C_SXP_SPEC_DATA	NA	
cb28	C_SXP_SELFI_DATA	NA	
cb30	C_SXP_TEXT_MASKDATA	NA	
cb32	C_SXP_TEXT2_MASKDATA	NA	
cb34	C_SXP_OPAC_MASKDATA	NA	
cb36	C_SXP_BUMP_MASKDATA	NA	
cb38	C_SXP_SPEC_MASKDATA	NA	
cb3a	C_SXP_SHIN_MASKDATA	NA	
cb3e	C_SXP_REFL_MASKDATA	NA	
cc00	C_NET_USE_VPOST	NA	
cc10	C_NET_USE_GAMMA	NA	
cc20	C_NET_FIELD_ORDER	NA	
cd00	C_BLUR_FRAMES	NA	
cd10	C_BLUR_SAMPLES	NA	
cd20	C_BLUR_DUR	NA	
cd30	C_HOT_METHOD	NA	
cd40	C_HOT_CHECK	NA	
cd50	C_PIXEL_SIZE	NA	
cd60	C_DISP_GAMMA	NA	
cd70	C_FBUF_GAMMA	NA	
cd80	C_FILE_OUT_GAMMA	NA	
cd82	C_FILE_IN_GAMMA	NA	

Ex.1007 / Page 328 of 1158
APPLE INC.

ID	Name	Data Type	Description
cd84	C_GAMMA_CORRECT	NA	
cd90	C_APPLY_DISP_GAMMA	NA	
cda0	C_APPLY_FBUF_GAMMA	NA	
cdb0	C_APPLY_FILE_GAMMA	NA	
cdc0	C_FORCE_WIRE	NA	
cdd0	C_RAY_SHADOWS	NA	
cde0	C_MASTER_AMBIENT	NA	
cdf0	C_SUPER_SAMPLE	NA	
ce00	C_OBJECT_MBUR	NA	
ce10	C_MBLUR_DITHER	NA	
ce20	C_DITHER_24	NA	
ce30	C_SUPER_BLACK	NA	
ce40	C_SAFE_FRAME	NA	
ce50	C_VIEW_PRES_RATIO	NA	
ce60	C_BGND_PRES_RATIO	NA	
ce70	C_NTH_SERIAL_NUM	NA	
d000	VPDATA	NA	
d100	P_QUEUE_ENTRY	NA	
d110	P_QUEUE_IMAGE	NA	
d114	P_QUEUE_USEIGAMMA	NA	
d120	P_QUEUE_PROC	NA	
d130	P_QUEUE_SOLID	NA	
d140	P_QUEUE_GRADIENT	NA	
d150	P_QUEUE_KF	NA	
d152	P_QUEUE_MOTBLUR	NA	
d153	P_QUEUE_MB_REPEAT	NA	
d160	P_QUEUE_NONE	NA	
d180	P_QUEUE_RESIZE	NA	
d185	P_QUEUE_OFFSET	NA	
d190	P_QUEUE_ALIGN	NA	
d1a0	P_CUSTOM_SIZE	NA	
d210	P_ALPH_NONE	NA	
d220	P_ALPH_PSEUDO	NA	
d221	P_ALPH_OP_PSEUDO	NA	
d222	P_ALPH_BLUR	NA	
d225	P_ALPH_PCOL	NA	
d230	P_ALPH_CO	NA	
d231	P_ALPH_OP_KEY	NA	
d235	P_ALPH_KCOL	NA	
d238	P_ALPH_OP_NOCONV	NA	

Ex.1007 / Page 329 of 1158
APPLE INC.

Autodesk 3D Studio (cont'd)

ID	Name	Data Type	Description
d240	P_ALPH_IMAGE	NA	
d250	P_ALPH_ALPHA	NA	
d260	P_ALPH_QUES	NA	
d265	P_ALPH_QUEIMG	NA	
d270	P_ALPH_CUTOFF	NA	
d280	P_ALPHANEG	NA	
d300	P_TRAN_NONE	NA	
d310	P_TRAN_IMAGE	NA	
d312	P_TRAN_FRAMES	NA	
d320	P_TRAN_FADEIN	NA	
d330	P_TRAN_FADEOUT	NA	
d340	P_TRANNEG	NA	
d400	P_RANGES	NA	
d500	P_PROC_DATA	NA	
f020	POS_TRACK_TAG_KEY	NA	
f021	ROT_TRACK_TAG_KEY	NA	
f022	SCL_TRACK_TAG_KEY	NA	
f023	FOV_TRACK_TAG_KEY	NA	
f024	ROLL_TRACK_TAG_KEY	NA	
f025	COL_TRACK_TAG_KEY	NA	
f026	MORPH_TRACK_TAG_KEY	NA	
f027	HOT_TRACK_TAG_KEY	NA	
f028	FALL_TRACK_TAG_KEY	NA	
f110	POINT_ARRAY_ENTRY	NA	
f120	POINT_FLAG_ARRAY_ENTRY	NA	
f130	FACE_ARRAY_ENTRY	NA	
f140	TEX_VERTS_ENTRY	NA	
f150	SMOOTH_GROUP_ENTRY	NA	
ffff	DUMMY	NA	

Chunk Order

Below is a list of data chunks, and beneath each one a list of chunks or subordinate chunks which usually follow them.

2200 (FOG)

COLOR_F fog_bgnd

2300 (DISTANCE_CUE)

dcue_bgnd *(data type unknown)*

Ex.1007 / Page 330 of 1158
APPLE INC.

1200 (SOLID_BGND)
    COLOR_F

1300 (V_GRADIENT)
    COLOR_F *(start of color gradient)*
    COLOR_F *(middle)*
    COLOR_F *(end of gradient)*

4600 (N_DIRECT_LIGHT)
    COLOR_F

7001 (VIEWPORT_LAYOUT)
    VIEWPORT_SIZE
    VIEWPORT_DATA

a010 (MAT_AMBIENT)
    COLOR_F *or* COLOR_24

a040 (MAT_SHININESS)
    percentage chunk 0030 *or* 0031

a200 (MAT_TEXMAP)
    percentage chunk
    MAT_MAPNAME
    MAT_MAP_TILING
    MAT_MAP_TEXBLUR
    possibly other material modifier chunks

a204 (MAT_SPECMAP)
    percentage chunk
    MAT_MAPNAME

b000 (KFDATA)
    KFHDR

b002 (OBJECT_NODE_TAG)
    NODE_HDR
    PIVOT
    POS_TRACK_TAG
    ROT_TRACK_TAG

Ex.1007 / Page 331 of 1158
APPLE INC.

SCL_TRACK_TAG

possibly others

b003 (CAMERA_NODE_TAG)

NODE_HDR

POS_TRACK_TAG

FOV_TRACK_TAG

ROLL_TRACK_TAG

possibly others

b004 (TARGET_NODE_TAG)

NODE_HDR

POS_TRACK_TAG

possibly others

b005 (LIGHT_NODE_TAG)

NODE_HDR

POS_TRACK_TAG

COL_TRACK_TAG

possibly others

b006 (L_TARGET_NODE_TAG)

NODE_ID

NODE_HDR

POS_TRACK_TAG

b007 (SPOTLIGHT_NODE_TAG)

NODE_ID

NODE_HDR

POS_TRACK_TAG

HOT_TRACK_TAG

FALL_TRACK_TAG

ROLL_TRACK_TAG

COL_TRACK_TAG

possibly others

Ex.1007 / Page 332 of 1158
APPLE INC.

b00a (KFHDR)

- VIEWPORT_LAYOUT
- KFSEG
- KFCURTIME
- OBJECT_NODE_TAG
- LIGHT_NODE_TAG
- TARGET_NODE_TAG
- CAMERA_NODE_TAG
- L_TARGET_NODE_TAG
- SPOTLIGHT_NODE_TAG
- AMBIENT_NODE_TAG
- possibly others

Chunk Dependencies

Chunk 4110 (POINT_ARRAY) relies on the following POINT definition:

```
typedef struct _POINT
{
   FLOAT x, y, z;
} POINT;
```

Chunk 4120 (FACE_ARRAY) relies on the following FACE definition:

```
typedef struct _FACE
{
   SHORT vertex1, vertex2, vertex3, flags;
} FACE;
```

Chunk 4140 (TEX_VERTS) relies on the following VERTEX definition:

```
typedef struct _VERTEX
{
   FLOAT x,y;
} VERTEX;
```

Chunk b020 (POS_TRACK_TAG) relies on the following POSITION definition:

```
typedef struct _POSITION
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT position_x, position_y, position_z;
} POSITION;
```


Ex.1007 / Page 333 of 1158
APPLE INC.

Chunk b021 (ROT_TRACK_TAG) relies on the following ROTATION definition:

```
typedef struct _ROTATION
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT rotation;          /* in radians */
   FLOAT axis_x, axis_y, axis_z;
} ROTATION;
```

Chunk b022 (SCL_TRACK_TAG) relies on the following definition of SCALE:

```
typedef _SCALE
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT scale_x, scale_y, scale_z;
} SCALE;
```

Chunk b023 (FOV_TRACK_TAG) relies on the following definition of FOV:

```
typedef _FOV
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT field_of_view;
} FOV
```

Chunk b024 (ROLL_TRACK_TAG) relies on the following ROLL definition:

```
typedef _ROLL
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT camera_roll;
} ROLL;
```

Chunk b025 (COL_TRACK_TAG) relies on the following definition of COLOR:

```
typedef _COLOR
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT r,g,b;
} COLOR;
```


Ex.1007 / Page 334 of 1158
APPLE INC.

Chunk b026 (MORPH_TRACK_TAG) relies on the following definition of MORPH:

```
typedef _MORPH
{
   SHORT frame_number;
   DWORD unknown;
   CHAR  object_name[]; /* 0-terminated ASCII string */
} MORPH;
```

Chunk b027 (HOT_TRACK_TAG) relies on the following definition of HOTSPOT:

```
typedef _HOTSPOT
{
   SHORT frame_number;
   DWORD unknown;
   FLOAT falloff_angle;
} HOTSPOT;
```

ASC File Example

This is a definition of a cube, centered at the origin, and saved in 3D Studio ASC format:

```
Ambient light color: Red=0.3 Green=0.3 Blue=0.3

Named object: "Cube"
Tri-mesh, Vertices: 8     Faces: 12
Vertex list:
Vertex 0: X:-1.000000     Y:-1.000000     Z:0.000000
Vertex 1: X:-1.000000     Y:-1.000000     Z:2.000000
Vertex 2: X:1.000000     Y:-1.000000     Z:0.000000
Vertex 3: X:1.000000     Y:-1.000000     Z:2.000000
Vertex 4: X:-1.000000     Y:1.000000     Z:0.000000
Vertex 5: X:1.000000     Y:1.000000     Z:0.000000
Vertex 6: X:1.000000     Y:1.000000     Z:2.000000
Vertex 7: X:-1.000000     Y:1.000000     Z:2.000000
Face list:
Face 0:   A:2 B:3 C:1 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 1:   A:2 B:1 C:0 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 2:   A:4 B:5 C:2 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 3:   A:4 B:2 C:0 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
```


Ex.1007 / Page 335 of 1158
APPLE INC.

```
Smoothing: 1
Face 4:    A:6 B:3 C:2 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 5:    A:6 B:2 C:5 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 6:    A:6 B:7 C:1 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 7:    A:6 B:1 C:3 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 8:    A:6 B:5 C:4 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 9:    A:6 B:4 C:7 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 10:    A:1 B:7 C:4 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
Face 11:    A:1 B:4 C:0 AB:1 BC:1 CA:1
Material:"r210g210b210a0"
Smoothing: 1
```

For Further Information

Autodesk sells a package that includes documentation on the 3DS file format. Please contact them for further details.

Autodesk Developer Marketing
2320 Marinship Way
Sausalito, CA 94965
Voice: 415-491-8719
WWW: *http://www.autodesk.com*

Information in this article was obtained from the following two documents:

Lewis, Jeff, *The Unofficial 3D Studio 3DS File Format*, v1.0, 1 December 1995.

Pitts, Jim, *3D Studio File Format (3dS)*, Document Revision 0.8, 18 December 1994.

Ex.1007 / Page 336 of 1158
APPLE INC.

These can be found at:

http://www.europa.com/~keithr/

Ex.1007 / Page 337 of 1158
APPLE INC.

BDF

NAME:	BDF
ALSO KNOWN AS:	Bitmap Distribution Format
TYPE:	Bitmap
COLORS:	Mono
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	Unlimited
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	ASCII
ORIGINATOR:	X Consortium
PLATFORM:	Any supporting X Window System
SUPPORTING APPLICATIONS:	Many X applications
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Used to store and exchange font information.

COMMENTS: Can be used to store arrays of ornamental glyph and other bitmap data.

Overview

BDF (Bitmap Distribution Format) is used by the X Window System as a method of storing and exchanging font data with other systems. The current version of BDF is 2.1; it is part of X11 Release 6. BDF is similar in concept to the PostScript Page Description Language. Both formats store data as printable ASCII characters, using lines of ASCII text that vary in length. Each line is terminated by an end-of-line character that may be a carriage return (ASCII 0Dh), a linefeed (ASCII 0Ah), or both.

Each BDF file stores information for exactly one typeface at one size and orientation (in other words, a font). A typeface is the name of the type style, such as Goudy, Courier, or Helvetica. A font is a variation in size, style, or orientation of a typeface, such as Goudy 10 Point, Courier Italic, or Helvetica Reversed. A glyph is a single character of a font, such as the letter "j". A BDF file therefore contains the data for one or more glyphs of a single font and typeface.

Ex.1007 / Page 338 of 1158
APPLE INC.

File Organization

A BDF file begins with information pertaining to the typeface as a whole, followed by the information on the font, and finally by the bitmapped glyph information itself. The information in a BDF file is stored in a series of records. Each record begins with an uppercase keyword, followed by one or more fields called tokens:

```
KEYWORD <token> <token> ...
```

All records, keywords, and information fields contain only ASCII characters and are separated by spaces. Lines are terminated by a <CR>, <LF>, or <CR/LF> pair. More than one record may appear on a physical line.

File Details

Following are some of the more common records found in BDF files:

```
STARTFONT <version>
ENDFONT
```

All BDF files begin with the STARTFONT record. This record contains a single information field indicating the version of the BDF format used in the file. The STARTFONT record contains all of the information within the BDF file and is terminated by the ENDFONT keyword as the last record in the file.

```
COMMENT <text>
```

COMMENT records may be found anywhere between the STARTFONT and ENDFONT records. They usually contain human-readable text that is ignored by font-reader applications.

```
FONT <fontname>
```

The FONT record specifies the name of the font contained within the BDF file. The name is specified using either the XFD font name or a private font name. The name may contain spaces, and the line containing the FONT record must be terminated by an end-of-line character.

```
SIZE <pointsize> <x resolution> <y resolution>
```

SIZE specifies the size of the font in points and the resolution of the output device that is to support the font.

```
FONTBOUNDINGBOX <width> <height> <x offset> <y offset>
```


Ex.1007 / Page 339 of 1158
APPLE INC.

The FONTBOUNDINGBOX record stores the size and the position of the font's bounding box as an offset from the origin (the lower-left corner of the bitmap).

```
STARTPROPERTIES <number of properties>
ENDPROPERTIES
```

The STARTPROPERTIES record contains subrecords that define the characteristics of the font. The STARTPROPERTIES keyword is followed by the number of properties defined within this record. The subrecords specify information such as the name of the font's creator, the typeface of the font, kerning and other rendering information, and copyright notices. The ENDPROPERTIES record always terminates the STARTPROPERTIES record. Following the ENDPROPERTIES record is the actual font data.

Following are descriptions of some common record keywords that may be used to describe the font data:

```
CHARS <number of segments>
```

The CHARS record indicates the number of font character (glyph) segments stored in the file.

```
STARTCHAR <glyphname>
ENDCHAR
```

The STARTCHAR record contains subrecords that store each glyph's information and data. The STARTCHAR keyword is followed by the name of the glyph. This name can be up to 14 characters in length and may not contain any spaces. The subrecords specify the index number of the glyph, the scalable width, and the position of the character.

The BITMAP record contains the actual glyph data encoded as 4-digit hexadecimal values. All bitmapped lines are padded on the right with zeros out to the nearest byte boundary. All of the glyph information is contained between the STARTCHAR record and the terminating ENDCHAR record. There is one STARTCHAR/ENDCHAR section per glyph stored in the BDF file.

Refer to the BDF documentation included with the X11R6 distribution for more information about the BDF information records.

Following is an example of a BDF file containing the characters `j` and quoteright (`\&'`). Note that more than one record appears per physical line:

```
STARTFONT 2.1 COMMENT This is a sample font in 2.1 format.
FONT -Adobe-Helvetica-Bold-R-Normal--24-240-75-75-
P-65-ISO8859-1 SIZE 24 75 75 FONTBOUNDINGBOX 9 24 -2 -6
```


Ex.1007 / Page 340 of 1158
APPLE INC.

```
STARTPROPERTIES 19 FOUNDRY "Adobe" FAMILY "Helvetica"
WEIGHT_NAME "Bold" SLANT "R" SETWIDTH_NAME "Normal"
ADD_STYLE_NAME "" PIXEL_SIZE 24 POINT_SIZE 240 RESOLUTION_X
75 RESOLUTION_Y 75 SPACING "P" AVERAGE_WIDTH 65
CHARSET_REGISTRY "ISO8859" CHARSET_ENCODING "1" MIN_SPACE 4
FONT_ASCENT 21 FONT_DESCENT 7 COPYRIGHT "Copyright (c) 1987
Adobe Systems, Inc." NOTICE "Helvetica is a registered
trademark of Linotype Inc." ENDPROPERTIES CHARS 2 STARTCHAR
j ENCODING 106 SWIDTH 355 0 DWIDTH 8 0 BBX 9 22 -2 -6 BITMAP
0380 0380 0380 0380 0000 0700 0700 0700 0700 0E00 0E00 0E00
0E00 0E00 1C00 1C00 1C00 1C00 3C00 7800 F000 E000 ENDCHAR
STARTCHAR quoteright ENCODING 39 SWIDTH 223 0 DWIDTH 5 0 BBX
4 6 2 12 ATTRIBUTES 01C0 BITMAP 70 70 70 60 E0 C0 ENDCHAR
ENDFONT
```

The following is the same BDF file with each of the records stored on separate lines and indented to illustrate the layering of BDF records and subrecords:

```
STARTFONT 2.1
    COMMENT This is a sample font in 2.1 format.
    FONT -Adobe-Helvetica-Bold-R-Normal--24-240-75-75-P-65-ISO8859-1
    SIZE 24 75 75
    FONTBOUNDINGBOX 9 24 -2 -6
    STARTPROPERTIES 19
        FOUNDRY "Adobe"
        FAMILY "Helvetica"
        WEIGHT_NAME "Bold"
        SLANT "R"
        SETWIDTH_NAME "Normal"
        ADD_STYLE_NAME ""
        PIXEL_SIZE 24
        POINT_SIZE 240
        RESOLUTION_X 75
        RESOLUTION_Y 75
        SPACING "P"
        AVERAGE_WIDTH 65
        CHARSET_REGISTRY "ISO8859"
        CHARSET_ENCODING "1"
        MIN_SPACE 4
        FONT_ASCENT 21
        FONT_DESCENT 7
        COPYRIGHT "Copyright (c) 1987  Adobe Systems, Inc."
        NOTICE "Helvetica is a registered trademark of Linotype Inc."
    ENDPROPERTIES
    CHARS 2
    STARTCHAR j
        ENCODING 106
        SWIDTH 355 0
        DWIDTH 8 0
        BBX 9 22 -2 -6
        BITMAP 0380 0380 0380 0380 0000 0700 0700 0700 0700 0E00 0E00
```


Ex.1007 / Page 341 of 1158
APPLE INC.

```
0E00 0E00 0E00 1C00 1C00 1C00 1C00 3C00 7800 F000 E000
    ENDCHAR
    STARTCHAR quoteright
        ENCODING 39
        SWIDTH 223 0
        DWIDTH 5 0
        BBX 4 6 2 12
        ATTRIBUTES 01C0
        BITMAP 70 70 70 60 E0 C0
    ENDCHAR
ENDFONT
```

For Further Information

For further information, see the BDF specification on the CD-ROM that accompanies this book. You may also find information about the BDF format in the X11R5 distribution of the X Window System, available via FTP from:

ftp://ftp.x.org.

Ex.1007 / Page 342 of 1158
APPLE INC.

NAME:	BRL-CAD
ALSO KNOWN AS:	Ballistic Research Laboratory CAD Package
TYPE:	See article
COLORS:	NA
COMPRESSION:	NA
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	See article
ORIGINATOR:	U.S. Army Advanced Communication Systems
PLATFORM:	UNIX
SUPPORTING APPLICATIONS:	BRL-CAD
SPECIFICATION ON CD:	Yes (summary only; specification is too lengthy)
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Solid modeling, network-distributed image processing.

COMMENTS: A massive, polymorphic system consisting of several standards.

Overview

BRL-CAD (Ballistic Research Laboratory CAD) is a solid-modeling system that originated at the Advanced Computing Systems Group of the U.S. Army Ballistic Research Laboratory. It was originally designed to provide an interactive editor for use in conjunction with a vehicle-description database. In the U.S. Army, "vehicle" often means "tank," and the documention contains many interesting and high-quality renderings of tank-like objects.

BRL-CAD is massive, consisting of more than 100 programs, and includes about 280,000 lines of C source code. It is extraordinarily well-documented and has been distributed to at least 800 sites worldwide.

Ex.1007 / Page 343 of 1158
APPLE INC.

Conceptually, BRL-CAD implements several subsystems:

- Solid geometry editor
- Ray tracer and ray tracing library
- Image-processing utilities
- General utilities

Data in the current release of BRL-CAD can be in several forms:

- BRL-specific CSG (Constructive Solid Geometry) database
- Uniform B-Spline and NURB surfaces
- Faceted data
- NMG (n-manifold geometry)

For Further Information

The BRL-CAD documentation is extensive and well-written, and we found it a pleasure to work with. Unfortunately, it is too extensive to be included on the CD, so we have elected to include only a summary description there. The full documentation is readily available in *The Ballistic Research Laboratory CAD Package*, Release 4.0, December 1991, albeit in paper form. Our copy came in a box weighing about ten pounds! It consists of five volumes:

Volume I	The BRL-CAD Philosophy
Volume II	The BRL-CAD User's Manual
Volume III	The BRL-CAD Applications Manual
Volume IV	The MGED User's Manual
Volume V	The BRL-CAD Analyst's Manual

This is an extraordinary document set, and not only in contrast to the rest of the documention we've run across in the research for this book. It's just a great job. If the application is one-tenth as well-crafted as the documentation, it must be a marvel.

For general information about BRL-CAD, contact:

Attn: Mike Muuss
BRL-CAD Architect
U.S. Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5068
mike@brl.mil

Ex.1007 / Page 344 of 1158
APPLE INC.

BRL-CAD is distributed in two forms:

1. Free distribution with no ongoing support. You must complete and return an agreement form; in return, you will be given instructions on how to obtain and decrypt the files via FTP. Files are archived at a number of sites worldwide. One copy of the printed documentation will be sent at no cost.

   For further information about this distribution, contact:

   BRL-CAD Distribution
   Attn: SCLBR-LV-V
   Aberdeen Proving Ground, MD 21005-5066
   FAX: 410-278-5058
   Email: *keith@brl.mil*

2. Full-service distribution with support. Items provided are similar to those mentioned in the free distribution described above, except that it costs U.S. $500 and may include the software on magnetic tape.

   For further information about this distribution, contact:

   BRL-CAD Distribution
   Attn: Mrs. Carla Moyer
   SURVIAC Aberdeen Satellite Office
   1003 Old Philadelphia Road, Suite 103
   Aberdeen, MD 21001
   Voice: 410-273-7794
   FAX: 410-272-6763
   Emial: *cad_dist@brl.mil*

For more information on BRL-CAD, see the following:

http://web.arl.mil/software/brlcad/index.html
BRL-CAD homepage

http://web.arl.mil/reports/
BRL-CAD technical reports

Ex.1007 / Page 345 of 1158
APPLE INC.

BUFR

NAME:	BUFR
ALSO KNOWN AS:	Binary Universal Form for the Representation of Meteorological Data
TYPE:	Various
COLORS:	NA
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	Binary bit-oriented
ORIGINATOR:	World Meteorological Organization
PLATFORM:	All
SUPPORTING APPLICATIONS:	Unknown
SPECIFICATION ON CD:	Yes (summary description)
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	GRIB

USAGE: Designed to convey meteorological data, it can be used for any other kind of data.

COMMENTS: The BUFR format is outside the scope of this book, but we include a brief description because it is likely to be more useful in the future as interest in geographical information systems increases.

Overview

BUFR (Binary Universal Form for the Representation of Meteorological Data) was created by the World Meteorological Organization (WMO). Technically it is known as WMO Code Form FM 94-IX Ext. BUFR. It is the result of a committee, which produced the first BUFR documents in 1988. The current revision of the format, Version 2, dates from 1991. Work on the format is ongoing. It is a code in the sense that it defines a protocol for the transmission of quantitative data, one of a number of codes created by the WMO.

BUFR was designed to convey generalized meteorological data, but due to its flexibility it can be used for almost anything. BUFR files, in fact, were designed

Ex.1007 / Page 346 of 1158
APPLE INC.

to be infinitely extensible, and to this end are written in a unique data description language.

We've included BUFR in this book because it can and has been used for transmission and exchange of graphics data, although that is not its primary purpose. It also is associated with observational data obtained from weather satellites.

BUFR data streams and files adhere to the specification called *WMO Standard Formats for Weather Data Exchange Among Automated Weather Information Systems.*

File Organization

BUFR files are stream-based and consist of a number of consecutive records. The format documentation describes BUFR records as self-descriptive. Records, or messages, make up the BUFR data stream, and each always contains a table consisting of a complete description of the data contained in the record, including data type identification, units, scaling, compression, and bits per data item.

File Details

Detailing the data definition language implemented in BUFR is beyond the scope of this article. It is extremely complex and is, at this point, used in a narrow area of technology.

For Further Information

For detailed information about BUFR, see the summary description included on the CD that accompanies this book:

> Thorpe, W., "Guide to the WMO Code Form FM 94-IX EXT. BUFR," Fleet Numerical Oceanography Center, Monterey, California.

Although there are a number of documents on BUFR available from meteorological sources, this article is the most useful that we have found. Additional information about WMO data specifications can be found in the following official specification:

> *Standard Formats for Weather Data Exchange Among Automated Weather Information Systems,* Document Number FCM-S2-1990.

Ex.1007 / Page 347 of 1158
APPLE INC.

This document is available from:

U.S. Department of Commerce/National Oceanic and Atmospheric Administration (NOAA)
Attn: Ms. Lena Loman
Office of the Federal Coordinator for Meteorological Services and Supporting Research (OFCM)
6010 Executive Blvd, Suite 900
Rockville, MD 20852
Voice: 301-443-8704

For further information about the BUFR format, contact:

U.S. Department of Commerce/National Oceanic and Atmospheric Administration (NOAA)
Attn: Dr. John D. Stackpole
Chief, Production Management Branch, Automation Division
National Meteorological Center
WINMC42, Room 307, WWB
5200 Auth Road
Camp Springs, MD 20746
Voice: 301-763-8115
FAX: 301-763-8381
Email: *jstack@sun1.wwb.noaa.gov*

You can get online information about BUFR at:

http://dao.gsfc.nasa.gov/data_stuff/formatPages/BUFR.html

Ex.1007 / Page 348 of 1158
APPLE INC.

NAME:	CALS Raster
ALSO KNOWN AS:	CALS, CAL, RAS
TYPE:	Bitmap
COLORS:	Mono
COMPRESSION:	CCITT Group 4, uncompressed
MAXIMUM IMAGE SIZE:	Unlimited
MULTIPLE IMAGES PER FILE:	Yes (Type II only)
NUMERICAL FORMAT:	NA
ORIGINATOR:	U.S. Department of Defense
PLATFORM:	All
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Compound document exchange, DTP, CAD/CAM, image processing.

COMMENTS: A well-documented, though cumbersome, format that attempts to do many things. If you are unfamiliar with U.S. government specification documents, you will probably find working with this format a complicated and challenging task. CALS raster is mandatory for use in most U.S. government document-handling applications. Because all data is byte-oriented, big-endian versus little-endian problems never arise.

Overview

The CALS raster format is a standard developed by the Computer Aided Acquisition and Logistics Support (CALS) office of the United States Department of Defense to standardize graphics data interchange for electronic publishing, especially in the areas of technical graphics, CAD/CAM, and image processing applications.

CALS is also an Office Document Interchange Format (ODIF) used in the Office Document Architecture (ODA) system for the exchange of compound document data between multiple machine platforms and software applications. CALS is an attempt to integrate text, graphics, and image data into a standard

Ex.1007 / Page 349 of 1158
APPLE INC.

document architecture. Its ultimate goal is to improve and integrate the logistics functions of the military and its contractors.

All technical publications for the federal government must conform to the CALS standard. Many other government organizations are also quickly adopting CALS. Commercial businesses, such as the medical, telecommunications, airline, and book publishing industries have also standardized on CALS.

CALS has also come into wide use in the commercial computer industry, such as in CAD/CAM applications, and in the aerospace industry, which owes a large part of its business to government and military contracts. CALS-compliant technical illustration systems also use the PostScript Page Description Language and Encapsulated PostScript files to exchange data between themselves and commercial systems.

File Organization

There are two types of CALS raster formats as defined by MIL-STD-28002A. They are specified as the Type I and Type II raster formats. Type I raster data files contain a single, monochrome image compressed using the CCITT Group 4 (T.6) encoding algorithm and appended to a CALS raster header record data block.

Type II image files contain one or more monochrome images that are also stored using the CCITT Group 4 encoding algorithm. In addition, the Type II format supports the encoding of image data as a collection of pel tiles. Each tile of image data is either separately encoded using CCITT Group 4 or is stored in its raw, unencoded format. The location of each tile within the image is stored in a tile offset index, for convenient retrieval of individual tiles. For further detail on the CALS Type II raster graphics format, refer to MIL-R-28002A.

The structures of the two CALS variants, Type I and Type II, are shown below.

The Type I file format consists of the following:

Header
Image Data

The Type II file format looks like this:

Header
Document Profile
Presentation Styles
Document Layout

Ex.1007 / Page 350 of 1158
APPLE INC.

Root Layout
  Layout Object Page 1
    Tile Index
    Image Data
  Layout Object Page 2
    Tile Index
    Image Data
  Layout Object Page N
    Tile Index
    Image Data

As you can see, the Type II format is considerably more complex than the Type I. Each Type II file may contain one or more pages of image data. There is also a considerable amount of page and document formatting data present in a Type II file. But by far the most common use of the Type II format is simply to store a collection of Type I CALS raster images in the same physical file. In such an arrangement, all the image pages are untiled, CCITT Group 4 encoded, and the profile, style, and layout information are omitted.

The raster data in a Type I file is always encoded using the CCITT Group 4 encoding method. CCITT Group 3 encoded and unencoded data is not supported. Type II files may contain tiles that are either CCITT Group 4 encoded or raw, unencoded data. Both raw and encoded tiles may occur in the same Type II CALS file and are always 512 pels in size. If the end of the image is reached before a tile is completely encoded, then this partial tile is completed by adding padding.

Two other types of tiles found in Type II images are null foreground and null background tiles. Null tiles are entirely white or entirely black, depending upon the designated background and foreground colors. They are actually pseudo-tiles that are not present in the image data and have no tile offset value.

Tile data is stored in the image data along the pel path (rows) and down the line progression (columns). Storage of randomly distributed tiles is possible, but discouraged. Tiles are normally encoded, unless the image data is so complex that the time required to encode the image is too great or unless very little reduction in the size of the data would result if the tile were encoded. The inclusion of unencoded data in a T.6-encoded data stream is not supported by the CALS raster format.

Ex.1007 / Page 351 of 1158
APPLE INC.

File Details

This section contains detailed information about the components of a CALS raster file.

Header Record Data Block

The CALS raster header is different from most other graphics file format headers in that it is composed entirely of 7-bit ASCII characters in a human-readable format. When most graphics image files are displayed as a text file, seemingly random garbage is displayed on the screen. Listing a CALS raster file, however, will reveal ASCII information which is quite understandable to a human reader. The unintelligible garbage following the header is the compressed image data.

The CALS raster data file header is 1408 bytes and is divided into eleven 128-byte records. Each record begins with a predefined 7-bit ASCII record identifier that is followed by a colon. The remaining part of the 128-byte record is the record information. If a record contains no information, the character string NONE is found in the record. The remainder of the bytes in each record contain space characters (ASCII 32) as filler. All data in the header block is byte-oriented, so no adjustments need to be made for byte order.

Following the last record in the header is 640 bytes of padding that rounds the header out to a full 2048 bytes in length. In fact, the raster image data always begins at offset 2048. Although this padding is not actually defined as part of the header, additional records added to future versions of the CALS header would be placed in this area. The byte value used for the padding is usually a space character (ASCII 20h), but any ASCII character can be used.

The structure for the CALS raster header block is shown below.

```
typedef struct _CalsHeader
{
CHAR  SourceDocId[128];          /* Source Document Identifier   */
CHAR  DestDocId[128];            /* Destination Document ID  */
CHAR  TextFileId[128];           /* Text File Identifier   */
CHAR  FigureId[128];             /* Table Identifier   */
CHAR  SourceGraph[128];          /* Source System Filename   */
CHAR  DocClass[128];             /* Data File Security Label   */
CHAR  RasterType[128];           /* Raster Data Type   */
CHAR  Orientation[128];          /* Raster Image Orientation   */
CHAR  PelCount[128];             /* Raster Image Pel Count   */
CHAR  Density[128];              /* Raster Image Density   */
CHAR  Notes[128];                /* Notes   */
CHAR  Padding[640];              /* Pad header out to 2048-bytes   */
} CALSHEAD;
```


Ex.1007 / Page 352 of 1158
APPLE INC.

Image record identifiers
Each record in a CALS raster file starts with a record identifier, which is a string of ASCII characters followed by a colon and a single space. Record data immediately follows the record identifier. If the record does not contain any relevant data, then the ASCII string NONE is written after the identifier.

SourceDocId:
SourceDocId starts with the source system document identifier (srcdocid). This record is used by the source system (the system on which the document was created) to identify the document to which the image is attached. This identifier can be a document title, publication number, or other similar information.

DestDocId:
DestDocId starts with the destination system document identifier (dstdocid). This record contains information used by the destination organization to identify the document to which the image is attached. This record may contain the document name, number or title, the drawing number, or other similar information.

TextFileId:
TextFileId starts with the text file identifier (txtfilid). This record contains a string indicating the document page that this image page contains. A code is usually found in this record that identifies the section of the document to which the image page belongs. Such codes may include:

COV	Cover or title page
LEP	List of effective pages
WRN	Warning pages
PRM	Promulgation pages
CHR	Change record
FOR	Forword or preface
TOC	Table of contents
LOI	Lists of illustrations and tables
SUM	Safety Summary
PTn	Part number n
CHn	Chapter number n
SEn	Section number n
APP-n	Appendix n

Ex.1007 / Page 353 of 1158
APPLE INC.

GLS	Glossary
INX	Index
FOV	Foldout section

FigureId:
FigureId starts with the figure or table identifier (figid). This is the number by which the image page figure is referenced. A sheet number is preceded by the ASCII string –S and followed by the drawing number. A foldout figure is preceded by the ASCII string –F and followed by the number of pages in the foldout.

SourceGraph:
SourceGraph starts with the source system graphics filename (srcgph). This record contains the name of the image file.

DocClass:
DocClass starts with the data file security label (doccls). This record identifies the security level and restrictions that apply to this image page and/or associated document.

RasterType:
RasterType starts with the raster data type (rtype). This is the format of raster image data that follows the header record data block in this file. This record contains the character 1 for Type I raster data and 2 for Type II raster data.

Orientation:
Orientation starts with the raster image orientation identifier (rorient). This record indicates the the proper visual orientation of the displayed image. This data is represented by two strings of three numeric characters separated by a comma. The first three characters are the direction of the pel path of the image page. Legal values are 0, 90, 180, and 270 representing the number of degrees the image was rotated clockwise from the origin when scanned. A page scanned normally has a pel path of 0 degrees, while an image scanned in upside-down has a pel path of 180 degrees.

The second three characters represent the direction of line progression of the document. Allowed values are 90 and 270 representing the number of degrees clockwise of the line progression from the pel path direction. A normal image has a line progression of 270, while a mirrored image has a line progression of 90.

Ex.1007 / Page 354 of 1158
APPLE INC.

PelCount:
PelCount starts with the raster image pel count identifier (rpelcnt). This record indicates the width of the image in pels and the length of the image in scan lines. This data is represented by two strings of six numeric characters separated by a comma. Typical values for this record are shown in Table CALS Raster-1.

TABLE CALS RASTER-1: *Typical CALS Raster Pel Count Values*

Drawing Size	Pels Per Line, Number of Lines
A	001728,002200
B	002240,003400
C	003456,004400
D	004416,006800
E	006848,008800
F	005632,008000

Density:
Density starts with the raster image density identifier (rdensity). This density is a single four-character numeric string representing the numerical density value of the image. This record may contain the values 200, 240, 300, 400, 600, or 1200 pels per inch, with 300 pels per inch being the default.

Notes:
Notes starts with the notes identifier (notes). This is a record used to contain miscellaneous information that is not applicable to any of the other records in the CALS raster file header.

Example

This section contains an example of a CALS raster file header data block created by a facsimile (FAX) software application. This image file contains a single page of a facsimile document received via a computer facsimile card and stored to disk as a CALS raster image file.

The source of the document is identified as FAX machine number one. The destination is an identification number used to index the image in a database. The ID number is constructed from the date the FAX was received, the order in which it was received (e.g., it was the third FAX received that day), and the

Ex.1007 / Page 355 of 1158
APPLE INC.

total number of pages. The text file identifier indicates that this file is page three of a seven-page facsimile document, for example.

The figure record is not needed, so the ASCII string NONE appears in this field. The source graphics filename contains the MS-DOS filename of the CALS raster file in which the page is stored. The remaining records indicate that the FAX document is unclassified, contains Type I CALS raster image data, has a normal orientation, and that the size and density of the image correspond to that of a standard facsimile page. The Notes field contains a time stamp showing when the FAX was actually received.

Please note that this is only one possible way that data may appear in a CALS header block. Most government and military software applications create CALS header blocks that are far more cryptic and confusing than this example. On the other hand, several CAD packages create simpler CALS headers.

Following is an example of a CALS header record data block:

```
srcdocid:   Fax machine #1
dstdocid:   910814-003.007
txtfilid:   003,007
figid:      NONE
srcgph:     F0814003.007
doccls:     Unclass
rtype:      1
rorient:    000,270
rpelcnt:    001728,002200
rdensity:   0200
notes:      Fri Aug 14 12:21:43 1991 PDT
```

For Further Information

Information about the CALS raster format is found primarily in the following military standards documents:

Automated Interchange of Technical Information, MIL-STD-1840A. This document contains a description of the header (called a header record data block) in the CALS format.

Requirements for Raster Graphics Representation in Binary Format, MIL-R-28002A. This document contains a description of the image data in the CALS format.

Ex.1007 / Page 356 of 1158
APPLE INC.

Also, see the CALS homepage at:

http://www.acq.osd.mil/cals/

The CALS raster file format is supported through the following office of the Department of Defense:

CALS Management Support Office (DCLSO)
Office of the Assistant Director for Telecommunications
and Information Systems
Headquarters Defense Logistics Agency
Cameron Station
Alexandria, VA 22314

The documents MIL-STD-1840 and MIL-R-28002A may be obtained from agencies that distribute military specifications standards, including the following:

Standardization Documents Ordering Desk
Building 4D
700 Robbins Avenue
Philadelphia, PA 19111-5094

Global Engineering Documents
2805 McGaw Avenue
Irvine, CA 92714

Voice: 800-854-7179
Voice: 714-261-1455

Useful and readily available periodical articles on CALS and ODA include the following:

Dawson, F., and F. Nielsen, "ODA and Document Interchange," *UNIX Review*, vol. 8, no. 3, 1990, p. 50.

Hobgood, A., "CALS Implementation—Still a Few Questions," *Advanced Imaging*, April 1990, pp. 24–25.

Ex.1007 / Page 357 of 1158
APPLE INC.

CGM

NAME:	CGM
ALSO KNOWN AS:	Computer Graphics Metafile
TYPE:	Metafile
COLORS:	Unlimited
COMPRESSION:	RLE, CCITT Group3 and Group4
MAXIMUM IMAGE SIZE:	Unlimited
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	NA
ORIGINATOR:	ANSI, ISO
PLATFORM:	All
SUPPORTING APPLICATIONS:	Too many to list
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Standardized platform-independent format used for the interchange of bitmap and vector data.

COMMENTS: CGM is a very feature-rich format which attempts to support the graphic needs of many general fields (graphic arts, technical illustration, cartography, visualization, electronic publishing, and so on). While the CGM format is rich in features (many graphical primitives and attributes), it is less complex than PostScript, produces much smaller (more compact) files, and allows the interchange of very sophisticated and artistic images. In fact, so many features are available to the software developer that a full implementation of CGM is considered by some to be quite difficult. Nevertheless, CGM use is spreading quickly.

Overview

CGM (Computer Graphics Metafile) was developed by experts working on committees under the auspices of the International Standards Organization (ISO) and the American Standards National Institute (ANSI). It was specifically designed as a common format for the platform-independent interchange of bitmap and vector data, and for use in conjunction with a variety of input and

Ex.1007 / Page 358 of 1158
APPLE INC.

output devices. Although CGM incorporates extensions designed to support bitmap (called raster in the CGM specification) data storage, files in CGM format are used primarily to store vector information. CGM files typically contain either bitmap or vector data, but rarely both.

The newest revision of CGM is the CGM:1992 standard, which defines three upwardly compatible levels of increasing capability and functionality. Version 1 is the original CGM:1987 standard, a collection of simple metafile primitives. Version 2 metafiles may contain Closed Figures (a filled primitive comprised of other primitives). Version 3 is for advanced applications, and its metafiles may contain Beziers, NURBS, parabolic and hyperbolic arcs, and the Tile Array compressed tiled raster primitive.

CGM uses three types of syntactical encoding formats. All CGM files contain data encoded using one of these three methods:

- Character-based, used to produce the smallest possible file size for ease of storage and speed of data transmission
- Binary encoded, which facilitates exchange and quick access by software applications
- Clear-text encoded, designed for human readability and ease of modification using an ASCII text editor

CGM is intended for the storage of graphics data only. It is sometimes (erroneously) thought to be a data transfer standard for CAD/CAM data, like IGES, or a 3D graphic object model data storage standard. However, CGM is quite suited for the interchange of renderings from CAD/CAM systems, but not for the storage of the engineering model data itself.

CGM supports and is used by the Graphical Kernel System (GKS) standard, but is something completely different. GKS, which is in fact an API graphics library specification, is often mistaken for a graphics file format. CGM has found a role on most platforms as a method for the transfer of graphics data between applications. Programs that support CGM include most business graphics and visualization packages and many word processing and CAD applications.

Vector primitives supported by Version 1 CGM metafiles include lines, polylines, arcs, circles, rectangles, ellipses, polygons, and text. Each primitive may have one or more attributes, including fill style (hatch pattern), line or edge color, size, type, and orientation. CGM supports bitmaps in the form of *cell arrays* and *tile arrays*. The logical raster primitives of CGM are device-independent.

Ex.1007 / Page 359 of 1158
APPLE INC.

A minor point, but one worth noting, is that the three flavors of encoding supported by CGM may not all be readable by all software applications that import CGM files. Despite the existence of a solid body of rules and encoding schemes, CGM files are not universally interchangeable.

Many CGM file-writing applications support different subsets of standard features, often leaving some features out that may be required by other CGM readers. Also, because CGM allows vendor-specific extensions, many (such as custom fills) have been added, making full CGM support by an application difficult.

The CGM:1987 standard included a "Minimum Recommended Capabilities" list to aid developers in implementing a CGM application capable of reading and writing CGM metafiles correctly. Unfortunately, some of the big manufacturers chose to ignore even these modest requirements. Therefore, because it is impossible to police everyone who implements CGM in an application, many incompatibilities do exist.

In an effort to improve compatibility, the CGM:1992 standard removed the "Minimum Recommended Capabilities" list in anticipation of the publication of the CGM Amendment 1, which defines more stringent conformance requirements and a "Model Profile," which could be considered a minimal useful implementation level. Amendment 1 is entitled "Conformance, Rules for Profiles, and the Model Profille." Amendment 2, "Application Structuring," in the publication process as we go to press, provides the ability to "tag," reference, and index collections of elements with application significance. Modeled after SGML, the new features provide a way to "object structure" a metafile. (A 3D metafile project is also underway.)

File Organization and Details

All CGM files start with the same identifier, the BEGIN METAFILE statement, but its actual appearance in the file depends on how the file is encoded. In clear-text encoding, the element is simply the ASCII string BEGMF. If the file is binary encoded, you must read in the first two bytes as a word; the most significant byte (MSB) is followed in the file by the least significant byte (LSB). Bits in this word provide the following information:

15–12: Element class
11–05: Element ID
04–00: Parameter list length

Ex.1007 / Page 360 of 1158
APPLE INC.

BEGIN METAFILE is a "Delimiter Element," making it class 0. The element ID within that class is 1. The parameter list length is variable, so it must be ANDed out when comparing. The bit pattern is then:

```
0 0 0 0 0 0 0 0 0 0 1 X X X X X
```

To check it, simply AND the word with 0XFFE0 and compare it with 0X0020. In reading the standard, we get the impression that it is actually legal to add padding characters (nulls) to the beginning of the file. We rather doubt that anyone would actually do this, but it may be appropriate to read in words until a non-zero word is read and compare this word. You can read in full words because all elements are constrained to start on a word boundary.

For Further Information

CGM is both an ANSI and an ISO standard and has been adopted by many countries, such as Australia, France, Germany, Japan, Korea, and the United Kingdom. The full ANSI designation of the current version of CGM is:

> *Information Processing Systems—Computer Graphics Metafile for the Storage and Transfer of Picture Description Information,* ANSI/ISO 8632-1992 (commonly called CGM: 1992).

Note that CGM:1992 is the current standard. Be careful not to obtain the earlier ANSI X3.122-1986 if you need the latest standard. This earlier document, CGM:1986, defining the Version 1 metafile, was superseded by ISO/IEC 8632:1992. ANSI adopted CGM:1992 without modification and replaced ANSI X3.122-1986 with it. The CGM standard is contained in four ISO standards documents:

ISO 8632-1 Part 1: Functional Specification
ISO 8632-2 Part 2: Character Encoding
ISO 8632-3 Part 3: Binary Encoding
ISO 8632-4 Part 4: Clear Text Encoding

These may be purchased from any of the following organizations:

International Standards Organization (ISO)
1 rue de Varembe
Case Postal 56
CH-1211 Geneva 20 Switzerland
Voice: +41 22 749 01 11
FAX: +41 22 733 34 30

Ex.1007 / Page 361 of 1158
APPLE INC.

American National Standards Institute (ANSI)
Sales Department
1430 Broadway
New York, NY, 10018
Voice: 212-642-4900

Canadian Standards Association (CSA)
Sales Group
178 Rexdale Blvd.
Rexdale, Ontario, M9W 1R3
Voice: 416-747-4044

Other countries also make the CGM specification available through their standards organizations; these include DIN (Germany), BSI (United Kingdom), AFNOR (France), and JIS (Japan).

The National Institute of Standards and Technology (NIST) has set up a Web page for CGM at:

http://speckle.ncsl.nist.gov/~lsr/cgm_std.htm

NIST has also set up a CGM Testing Service for testing CGM metafiles, generators, and interpreters. The Testing Service examines binary-encoded CGM files for conformance to Version 1 CGM, as defined in the application profiles of FIPS 128-1 and the DoD CALS CGM AP military specification MIL-D-28003A. You can purchase the testing tool used by NIST so you can do internal testing on various PC and UNIX systems.

For more information about the CGM Testing Service, contact:

National Institute of Standards and Technology (NIST)
Computer Systems Laboratory
Information Systems Engineering Division
Gaithersburg, MD 20899
Voice: 301-975-3265

You can also obtain information about CGM from the following references:

Arnold, D.B. and P.R. Bono, *CGM and CGI: Metafile and Interface Standards for Computer Graphics*, Springer-Verlag, New York, NY, 1988.

Arnold, D.B. and P.R. Bono, *CGM et CGI: normes de metafichier et d'interfaces pour l'infographie*, French translation and updating of the above reference, Masson, 1992.

Ex.1007 / Page 362 of 1158
APPLE INC.

Henderson, L.R., and Gebhardt, "CGM: SGML for Graphics," *The Gilbane Report*, Fall 1994.

Henderson, L.R., and A.M. Mumford, *The CGM Handbook*, Academic Press, San Diego, CA, 1993.

Bono, P.R. , J.L. Encarnacao, L.M. Encarnacao, and W.R. Herzner, *PC Graphics With GKS*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

There are also two amendments to theis specification:

- Amendment 1. Conformance, Rules for Profiles, and Model Profile.
- Amendment 2. Application Structuring.

For additional information online, see:

http://www.agocg.ac.uk:80/agocg/cgm.html

Ex.1007 / Page 363 of 1158
APPLE INC.

CMU Formats

NAME:	CMU Formats
ALSO KNOWN AS:	Andrew Formats, CMU Bitmap
TYPE:	Multimedia
COLORS:	NA
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	NA
ORIGINATOR:	Carnegie Mellon University
PLATFORM:	All
SUPPORTING APPLICATIONS:	Andrew Toolkit
SPECIFICATION ON CD:	Yes
CODE ON CD:	NO
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Used primarily at Carnegie Mellon University in conjunction with the Andrew Toolkit.

COMMENTS: Included mainly for its architectural uniqueness.

Overview

The Andrew Consortium at Carnegie Mellon University is the source of the Andrew Toolkit, which is associated with the Andrew User Interface System. The Toolkit API is the basis for applications in the Andrew User Interface System. Data objects manipulated by the Andrew Toolkit must adhere to conventions crystallized in the Andrew Data Stream specification, a draft of which is included on the CD accompanying this book. The system was designed to support multimedia data from a variety of programs and platforms.

We understand that there is a bitmap format which originated at Carnegie Mellon University, but were unable to locate information prior to publication. The PBM utilities may include some support for converting and manipulating a CMU Bitmap, however.

Ex.1007 / Page 364 of 1158
APPLE INC.

File Organization

In the CMU formats, data is organized into streams and is written in 7-bit ASCII text. This is an interesting idea—nearly unique in the graphics file format world—which appears designed to enhance the portability of the format, at some cost in file size. Text may include tabs and newline characters and is limited to 80 characters of data per line.

Note that Andrew Toolkit files assume access by the user to the Andrew Toolkit. In the words of the documentation authors:

> As usual in ATK, the appropriate way to read or write the data stream is to call upon the corresponding Read or Write method from the AUIS distribution. Only in this way is your code likely to continue to work in the face of changes to the data stream definition. Moreover, there are a number of special features—mostly outdated data streams—that are implemented in the code, but not described here.

File Details

Data files used by the Andrew Toolkit consist of data objects, which are marked in the file by a begin/end marker pair. The initial marker associated with each data object must include information denoting the object type, as well as a unique identifier, which may be used as a reference ID by other objects.

The following is an example from the documentation:

```
\begindata{text,1}
<text data>
\begindata{picture,2}
<picture data>
\enddata{picture,2}
\view {pictureview,2}
<more text data>
\enddata{text,1}
```

Text Data Streams

Text data streams are similar to other data streams. Their structure is as follows:

```
\begindata line
\textdsversion line
\template line
definitions of additional styles
the text body itself
```


Ex.1007 / Page 365 of 1158
APPLE INC.

```
styled text
embedded objects in text body
\enddata line
```

Each of these elements is described below.

\begindata
This line has the form:

```
\begindata{text,99999}
```

where 99999 is a unique identifier.

\textdsversion
This line has the form:

```
\textdsversion{12}
```

There are apparently files written with data stream versions other than 12.

\template
A file may use a style template, in which case there will be a line of the form:

```
\template{default}
```

where default is the name of the template used and is the prefix of a filename. The system appends a suffix .tpl and looks for the template along file paths defined in the Andrew Toolkit installation. Please see the specification for further information.

Definitions of additional styles
Additional styles may be defined and used on the fly; each style consists of two or more lines:

```
\define{internalstylename
menuname
attribute
. . .
attribute}
```

internalstylename is always written in lowercase and may not contain spaces. The menuname line is optional. If it is missing, there must be an empty line in its place. If present, it has the form:

```
menu:[Menu card name,Style name]
```


Ex.1007 / Page 366 of 1158
APPLE INC.

Attributes are also optional; if they are missing, the closing } appears at the end of the menuname line. Attribute lines are of the form:

```
attr:[attributename basis units value]
```

where value is a signed integer.

Text body
Text consists of any number of consecutive lines, each terminated by a newline character.

Styled text
Text in the body may be displayed in a style, in which case it is preceded by a previously defined name:

```
\internalstylename{
```

and is followed by the corresponding closing brace.

Embedded objects
Objects may be embedded in the text body. The documentation for the CMU formats describes the use of embedded objects as follows:

> When an object is embedded in a text body, two items appear: the data stream for the object and a \view line. The \begindata for the object is always at the beginning of a line. (The previous line is terminated with a backslash if there is to be no space before the object.) The \enddata line for the object always ends with a newline (which is not treated as a space).
>
> The \view line has the form:
>
> ```
> \view{rasterview,8888,777,0,0}
> ```

\enddata
The \enddata line has the form:

```
\enddata{text,99999}
```

Bitmap Images

A bitmap image is a standard data stream beginning with a \begindata line and ending with a \enddata line. These generally surround a header and an image body.

Ex.1007 / Page 367 of 1158
APPLE INC.

The first line of the header consists of the following:

```
2 0 65536 65536 0 0 484 603
```

The following describes the numbers in this header:

Raster version	2	Denotes the second version of this encoding

Options	0	This field may specify changes to the image before displaying it:
		`raster_INVERT(1>>0) /* exch black & white */`
		`raster_FLIP(1>>1) /* exch top & bottom */`
		`raster_FLOP(1>>2) /* exch left & right */`
		`raster_ROTATE(1>>3) /* rotate 90 clockwise */`

xScale, yScale:	65536 65536	Affects the size at which the image is printed. The value raster_UNITSCALE (136535) prints the image at approximately the size on the screen. The default scale of 65,536 is approximately half the screen size.
x, y, width, height:	0 0 484 603	It is possible for a raster object to display a portion of an image. These fields select this portion by specifying the index of the upper-left pixel and the width and height of the image in pixels. In all instances so far, x and y are both zero, and the width and height specify the entire image.

The second header line has three possible variations. Currently, only the first is used.

Variation 1: bits 10156544 484 603

RasterType: bits
RasterId: 10156544
Width, Height: 484 603

Width and Height describe the width of each row and the number of rows.

Variation 2: refer 10135624

RasterType: refer
RasterId: 10135624

Ex.1007 / Page 368 of 1158
APPLE INC.

The current data object refers to the bits stored in another data object that appears earlier in the same data stream.

Variation 3: file 10235498 *filename path*
RasterType: file
RasterId: 10235498

The bit data is found in the file *filename.*

Please check the specification document on the CD-ROM for subtleties and further details of the format.

For Further Information

For further information about the CMU formats and the Andrew Toolkit, as well as the full Andrew source and binary code, other contributed software, and documentation, see the following Andrew Consortium online sites:

http://www.cs.cmu.edu/afs/cs.cmu.ed/project/atk-ftp/web/andrew-home.html
ftp://ftp.andrew.cmu.edu/pub/AUIS/

You can also contact:

Andrew Consortium
Attn: Ann Marie Zanger, Assistant Director
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890
Voice: 412-268-6710
Email: *info-andrew+@andrew.cmu.edu*

Ex.1007 / Page 369 of 1158
APPLE INC.

DKB

NAME:	DKB
ALSO KNOWN AS:	None
TYPE:	Scene format
COLORS:	NA
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	NA
ORIGINATOR:	David K. Buck
PLATFORM:	All
SUPPORTING APPLICATIONS:	DKB Ray Trace application
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	POV

USAGE: Designed to support the widely-distributed, but now obsolete, DKB ray trace application.

COMMENTS: The DKB ray tracer preceded, and has influenced, the development of the POV ray tracer.

Overview

The DKB ray trace application was created by David K. Buck, for whom it is named. DKB has enjoyed wide distribution, particularly in the PC/MS-DOS BBS world.

Mr. Buck declined our request for information on the format and permission to reprint the relevant documentation. He feels that there is no interest in the program, and that it has been superseded by the POV ray tracer, which is based in part on DKB. We think otherwise, of course. Once a format is out there—especially if it is distributed as freeware or shareware—it is out there, and there is no practical way to stop people from using it. Our survey of some of the large BBSs in the U.S. showed that DKB, along with other, older ray trace programs, is being downloaded with some regularity, despite the availability of

Ex.1007 / Page 370 of 1158
APPLE INC.

POV. Unfortunately, we are not able to describe the DKB format in any detail here.

For Further Information

The DKB ray trace package is available for download from Internet archive sites and BBS systems running on a number of platforms.

Ex.1007 / Page 371 of 1158
APPLE INC.

Dore Raster File Format

NAME:	Dore Raster File Format
ALSO KNOWN AS:	Dore, RFF
TYPE:	Bitmap
COLORS:	Unlimited
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	Unlimited
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Any
ORIGINATOR:	Kubota Pacific Computers
PLATFORM:	UNIX
SUPPORTING APPLICATIONS:	AVS visualization package, many others
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Storage and interchange of 2D and 3D image data.

COMMENTS: The Dore Raster File Format is one of the rare formats that combines both ASCII and binary information within the same image file. Dore lacks a native method of data compression, but does support the storage of voxel data.

Overview

The Dore Raster File Format (Dore RFF) is used by the Dore graphics library for storing 2D and 3D image data. Once stored, the data may be retrieved or the RFF files may be used to interchange the image information with other software packages.

Dore itself is an object-oriented 3D graphics library used for rendering 2D and 3D near-photographic quality images and scenes. Dore supports features such as texture and environment mapping, 2D and 3D filtering, and storage of 3D raster data.

Most raster (bitmap) file formats are only capable of storing 2D pixels (picture elements). Dore RFF is capable of storing 3D pixels called voxels (volume

Ex.1007 / Page 372 of 1158
APPLE INC.

elements). A voxel contains standard pixel information, such as color and alpha channel values, plus Z-depth information, which describes the relative distance of the voxel from a point of reference. The Dore RFF format defines Z-depth data as a 32-bit integer value in the range of 00h to FFFFFFFFh (2^{32} - 1). A value of 00h places a voxel at the farthest possible point from the point of reference, and a value of FFFFFFFFh places a voxel at the closest possible point.

The Dore Raster File Format is the underlying API for the AVS visualization package.

File Organization

Dore RFF is a rather simple, straightforward format that contains primarily byte-oriented data. All Dore RFF files contain an ASCII header, followed by binary image data.

Following is an example of a Dore RFF file. The position of the binary image data is indicated by the label in brackets. The <FF> symbols are formfeed (ASCII 0Ch) characters:

```
# # Dore Raster File Example
#
  rastertype = image
  width = 1280
  height = 1024
  depth = 2
  pixel = r8g8b8a8z32
  wordbyteorder = big-endian
<FF><FF>
[Binary Image Data]
<EOF>
```

The header may contain up to six fields. Each field has the format *keyword* = *value* and is composed entirely of ASCII characters. There is typically one field per line, although multiple fields may appear on a single line if they are separated by one or more white-space characters. Comments begin with the # character and continue to the end of the line.

Ex.1007 / Page 373 of 1158
APPLE INC.

The header fields are summarized below:

rastertype Type of raster image contained in the file. The only supported value for this keyword is image. This field must appear first in all RFF headers and has no default value.

width 2-byte WORD value that indicates the width of the image in pixels or voxels. The width field must appear in all RFF headers and has no default value.

height 2-byte WORD value that indicates the height of the image in pixels or voxels. The height field must appear in all RFF headers and has no default value.

depth 2-byte WORD value that indicates the depth of the raster. 2D raster images always contain pixels, and therefore the depth value is 1. 3D raster images always contain voxels, and therefore the depth value is always greater than 1. The depth field is optional and has a default value of 1.

pixel String indicating the data format of each pixel or voxel. The pixel field has no default value and must appear in all RFF headers. The possible values for this field are as follows:

- r8g8b8 Pixels are three bytes in size and stored as three separate 8-bit RGB values.
- r8g8b8a8 Pixels are four bytes in size and stored as three separate 8-bit RGB values and a single 8-bit alpha channel value in RGBA order.
- a8b8g8r8 Pixels are four bytes in size and stored as three separate 8-bit RGB values and a single 8-bit alpha channel value in ABGR order.
- a8g8b8a8z32 Voxels are eight bytes in size and stored as three separate 8-bit RGB values, a single 8-bit alpha channel value, and a single 32-bit DWORD containing the Z-depth value in RGBAZ order.

Ex.1007 / Page 374 of 1158
APPLE INC.

a8 Pixels or voxels are a single byte in size and only contain a single 8-bit alpha channel value.

z32 Voxels are a single byte in size and only contain a single 32-bit Z-depth value.

wordbyteorder

Indicates the byte-order of the Z-depth values in the binary image data. The value of this field may be either the string big-endian or little-endian. The appearance of the wordbyteorder field in the header is optional, and the value defaults to big-endian.

The order of the fields within the header is not significant, except for the rastertype field. rastertype must always appear in every RFF header and must always be the first field. The width, height, and pixel fields must appear in every header as well. All other fields are optional; if not present, their default values are assumed.

The End Of Header marker is two formfeed characters. If it is necessary to pad out the header to end on a particular byte boundary, you can place any number of ASCII characters, except formfeeds, between these two formfeed characters. See the example below:

```
# # Example of using the End Of Header marker to pad out the header
# # to a byte boundary
#
  rastertype = image
  width = 64
  height = 64
  depth = 1
  pixel = r8g8b8
  wordbyteorder = little-endian
<FF>
ASCII data used to extend the length of the header
<FF>
Binary Image Data
<EOF>
```

File Details

The image data immediately follows the End OF Header maker and is always stored in an uncompressed binary form. The format of the image data depends upon the format of the pixel (or voxel) data indicated by the value of the pixel field in the header.

Ex.1007 / Page 375 of 1158
APPLE INC.

The image data is always stored as contiguous pixels or voxels organized within scan lines. All RGB and alpha channel values are stored as bytes. Z-depth values are stored as 32-bit DWORDs, with the byte-order of the Z-depth values being indicated by the value of the wordbyteorder field in the header. Any pixel containing a Z-depth value is regarded as a voxel.

The format of each possible type of pixel or voxel is illustrated using C language structures as follows:

```
/* Pixel - Simple RGB triple */
typedef struct _r8g8b8
{
  BYTE red;
  BYTE green;
  BYTE blue;
} R8G8B8;
/* Pixel - RGB triple with alpha channel value */
typedef struct _r8g8b8a8
{
  BYTE red;
  BYTE green;
  BYTE blue;
  BYTE alpha;
} R8G8B8A8;
/* Pixel - RGB triple with alpha in reverse order */
typedef struct _a8b8g8r8
{
  BYTE alpha;
  BYTE blue;
  BYTE green;
  BYTE red;
} A8B8G8R8;
/* Voxel - RGB triple, alpha, and Z-depth */
typedef struct _a8g8b8a8z32
{
  BYTE red;
  BYTE green;
  BYTE blue;
  BYTE alpha
  DWORD zdepth;
} R8G8B8A8Z32;
/* Pixel or voxel mask - Only an alpha channel value */
typedef struct _a8 {
        BYTE alpha;
} A8;
/* Voxel mask - Only a Z-depth value */
typedef struct _z32
{
  DWORD zdepth;
} Z32;
```


Ex.1007 / Page 376 of 1158
APPLE INC.

For Further Information

For further information about the Dore Raster File Format, see the specification included on the CD-ROM that accompanies this book. You can also contact:

Kubota Pacific Computer, Inc.
Attn: Steve Hollasch
2630 Walsh Avenue
Santa Clara, CA 95051
Voice: 408-727-8100
Email: *hollasch@kpc.com*

You can also contact:

Lori Whippler
Email: *loriw@kpc.com*

Ex.1007 / Page 377 of 1158
APPLE INC.

DPX

NAME:	DPX
ALSO KNOWN AS:	SMPTE Digital Picture Exchange Format, SMPTE DPX, Kodak Cineon
TYPE:	Bitmap
COLORS:	1-, 8-, 10-, 12-, 16-, 32-, and 64-bit
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	4G×4G
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian or little-endian
ORIGINATOR:	Kodak
PLATFORM:	Any
SUPPORTING APPLICATIONS:	Kodak imaging applications
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Storage of digital motion picture data in a bitmap file format.

COMMENTS: DPX is a modification of the Kodak Cineon format. The Society of Motion Picture and Television Engineers (SMPTE) added header information to DPX not present in the original Kodak Cineon format.

Overview

DPX is a bitmap file format used to store a single frame of a motion picture or video data stream. Multiple DPX files are used to store and exchange digital moving picture sequences between a wide variety of electronic and computer systems.

The DPX format is an ANSI and SMPTE standard based on the Kodak Cineon file format. These two formats are nearly identical, except for several extra header fields defined in the DPX format. For this reason, we recommend that you support the DPX format even if you are just interested in parsing Cineon image files.

Ex.1007 / Page 378 of 1158
APPLE INC.

DPX has several features designed to support device and resolution independence. For example, DPX files, like TIFF files, may be written using either the big- or little-endian byte-ordering schemes. Image data is stored as a series of up to eight elements using one of the many color component formats supported by the DPX standard. Pixel data itself may be stored a depth of 1, 8, 10, 12, 16, 32, or 64 bits. Padding of scan lines and image data, and the packing of pixel data, is also supported.

File Organization

DPX files are organized as a series of headers followed by an optional block of user-defined data and finally by the image data itself. The DPX standard defines two headers:

- The generic header containing information on the DPX file, the image data, and the image orientation
- The industry-specific header, containing information used by the film and television industries

For clarity, this article will regard these sections as five separate headers.

Following these headers is an optional area containing user-defined data. This area may contain any type of information the DPX file writer needs to store this data and may be up to one megabyte in size.

Last in the file is the image data itself, stored as an array of pixel component values. Figure DPX-1 illustrates the internal arrangement of a DPX file.

File Details

The five DPX headers are all fixed in size and contain fields ranging in length from one to several hundred bytes. Fields of the SINGLE (16-bit) and DOUBLE (32-bit) floating-point data types are also common.

Any integer or real header fields that contain undefined data are to be initialized to their "all ones" value. In C, this is most easily accomplished by assigning the one's-complement of 0 to an undefined field, as follows:

```
WORD val = ~0;     /* Set val to 0xFFFF */
```

Undefined ASCII fields are initialized to all NULL (ASCII 00h) characters.

Ex.1007 / Page 379 of 1158
APPLE INC.

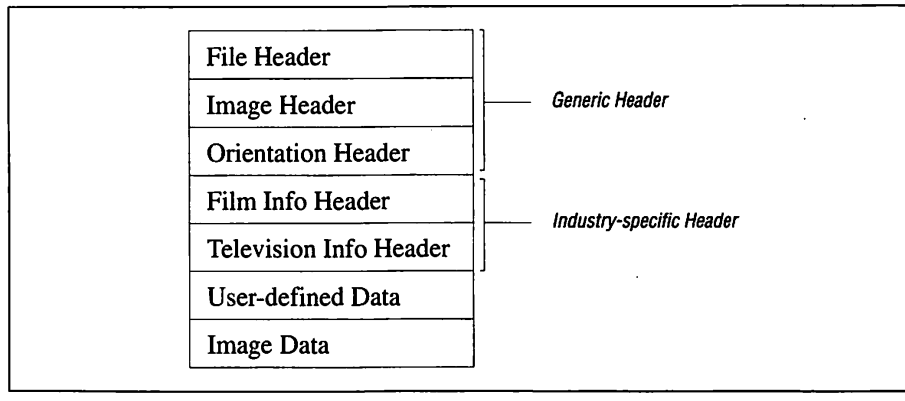


FIGURE DPX-1: *DPX format*

File Header

The file header contains information on the DPX file and all of its parts, including the name, size, and date stamp of the file, and the internal location of the image data. The DPX file header is 768 bytes in length and has the following format:

```
typedef struct _GenericFileHeader
{
    DWORD Magic;            /* Magic number */
    DWORD ImageOffset;      /* Offset to start of image data in bytes */
    char  Version[8];       /* Version stamp of header format */
    DWORD FileSize;         /* Total DPX file size in bytes */
    DWORD DittoKey;         /* Image content specifier */
    DWORD GenericSize;      /* Generic section header length in bytes */
    DWORD IndustrySize;     /* Industry-specific header length in bytes */
    DWORD UserSize;         /* User-defined data length in bytes */
    char  FileName[100];    /* Name of DPX file */
    char  TimeDate[24];     /* Time and date of file creation */
    char  Creator[100];     /* Name of file creator */
    char  Project[200];     /* Name of project */
    char  Copyright[200];   /* File contents copyright information */
    DWORD EncryptKey;       /* Encryption key */
    char  Reserved[104];    /* Reserved field used for padding */
} GENERICFILEHEADER;
```

Magic is the DPX file identification value. It is also the indicator of the byte order of the DPX file data. If this value is 53445058h ("SDPX"), the file data is written in big-endian order. If this value is 58504453h ("XPDS"), the file data is written in little-endian order.

Ex.1007 / Page 380 of 1158
APPLE INC.

ImageOffset is the offset of the start of the image data from the beginning of the file in bytes.

Version is an 8-byte ASCII field containing a string of the version of the DPX standard to which the file conforms. For v1.0 files, this field contains the ASCII string "V1.0".

FileSize is the total size of the DPX file in bytes.

DittoKey indicates whether the image in the DPX file is the same as in the previous frame (a value of 0), or is a new frame (a value of 1).

GenericSize is the total size of the generic header section in bytes.

IndustrySize is the total size of the industry-specific header section in bytes.

UserSize is the total size of the user-defined data area in bytes. If no user-defined area is present, its value will be 0.

FileName is a 100-byte ASCII field containing free-form text information identifying the name and path of the DPX file.

TimeDate is a 24-byte field containing a time and date stamp of when the DPX file was created. This stamp is a string in the form *YYYY:MM:DD:HH:MM:SS:LTZ*.

Creator is a 100-byte ASCII field containing free-form text information identifying the creator of the DPX file and/or the image it contains.

Project is a 200-byte ASCII string containing free-form text information identifying the project with which the DPX file and/or image is associated.

Copyright is a 200-byte ASCII string containing free-form text information identifying the copyright of the image data stored within the DPX file.

EncryptKey contains a 4-byte value used as a key to process the file using a user-specified decryption algorithm. EncryptKey has a value of FFFFFFFFh if the image data is not encrypted. The DPX header data is never encrypted.

Reserved is a 104-byte field used to pad the header out to 768 bytes in length. Future versions of the DPX format may support additional fields in this area.

Image Header

Immediately following the file header is the image header. This header stores information specific to the image data of the frame stored in the DPX file.

Ex.1007 / Page 381 of 1158
APPLE INC.

Note that this header contains a sequence of eight identical subheaders. Each subheader stores information on each of the eight possible elements that make up the image data.

The image header is 640 bytes in size and has the following format:

```
typedef struct _GenericImageHeader
{
    WORD  Orientation;            /* Image orientation */
    WORD  NumberElements;         /* Number of image elements */
    DWORD PixelsPerLine;          /* Pixels per line */
    DWORD LinesPerElement;        /* Lines per image element */
    struct _ImageElement
    {
        DWORD  DataSign;          /* Data sign extension */
        DWORD  LowData;           /* Reference low data code value */
        SINGLE LowQuantity;       /* Reference low quantity represented */
        DWORD  HighData;          /* Reference high data code value */
        SINGLE HighQuantity;      /* Reference high quantity represented */
        BYTE   Descriptor;        /* Descriptor for image element */
        BYTE   Transfer;          /* Transfer characteristics for element */
        BYTE   Colorimetric;      /* Colormetric specification for element */
        BYTE   BitSize;           /* Bit size for element */
        WORD   Packing;           /* Packing for element */
        WORD   Encoding;          /* Encoding for element */
        DWORD  DataOffset;        /* Offset to data of element */
        DWORD  EndOfLinePadding;  /* End of line padding used in element */
        DWORD  EndOfImagePadding; /* End of image padding used in element */
        char   Description[32];   /* Description of element */
    } ImageElement[8];
    BYTE Reserved[52];            /* Reserved field used for padding */
} GENERICIMAGEHEADER;
```

Orientation specifies the proper direction of the image on the display. Line direction and frame direction are specified, respectively, by the values:

0 Left to right, top to bottom
1 Right to left, top to bottom
2 Left to right, bottom to top
3 Right to left, bottom to top
4 Top to bottom, left to right
5 Top to bottom, right to left
6 Bottom to top, left to right
7 Bottom to top, right to left

0 is the only value supported for the core set file format.

Ex.1007 / Page 382 of 1158
APPLE INC.

NumberElements indicates the number of elements in the image data. This value also indicates the number of elements in the ImageElement[] array that contain valid data.

PixelsPerLine specifies the number of pixels in a scan line. There is an equal number of pixels in each line per element.

LinesPerElement specifies the number of lines in the element. There is an equal number of lines in each element.

The following 15 fields are repeated eight times in the generic image header. Each of these 72-byte subheaders contain information on a single element of the image data. The fields of all eight subheaders will always be present in each DPX file, even if there is only one element of image data.

DataSign is 0 if the image data is unsigned and 1 if the data is signed. Image data is normally stored using unsigned values.

LowData defines the minimum expected color value.

LowQuantity defines the reference for the low quantity value specified in the LowData field. This reference value should be considered the lowest possible value and is typically 0.

HighData defines the maximum expected color value.

HighQuantity defines the reference for the high-quantity value specified in the HighData field. This reference value should be considered the highest possible value, and is typically the largest value that can be stored in a component of the specified size (for example, 256 for 8-bit components).

Descriptor specifies the type of component stored by the element and its pixel-packing order. There are 256 possible values to this field; the following are defined:

0	User-defined
1	Red
2	Green
3	Blue
4	Alpha
6	Luminance
7	Chrominance
8	Depth
9	Composite video

Ex.1007 / Page 383 of 1158
APPLE INC.

50	RGB
51	RGBA
52	ABGR
100	CbYCrY
101	CbYaCrYa
102	CbYCr
103	CbYCra
150	User-defined 2-component element
151	User-defined 3-component element
152	User-defined 4-component element
153	User-defined 5-component element
154	User-defined 6-component element
155	User-defined 7-component element
156	User-defined 8-component element

All values not defined are reserved for use in future versions of DPX.

Transfer describes the transfer characteristics used to transform the data from a linear original. The possible values of this field are:

0	User-defined
1	Printing density
2	Linear
3	Logarithmic
4	Unspecified video
5	SMPTE 240M
6	CCIR 709-1
7	CCIR 601-2 system B or G
8	CCIR 601-2 system M
9	NTSC composite video
10	PAL composite video
11	Z linear
12	Z homogeneous

Colorimetric defines the color reference primaries for color additive systems, or color responses for color subtractive systems. The possible values of this field are:

Ex.1007 / Page 384 of 1158
APPLE INC.

0 User-defined
1 Printing density
4 Unspecified video
5 SMPTE 240M
6 CCIR 709-1
7 CCIR 601-2 system B or G
8 CCIR 601-2 system M
9 NTSC composite video
10 PAL composite video

BitSize specifies the number of bits in each component of the image element. Valid values are 1, 8, 10, 12, 16, 32, and 64.

Packing defines the method used to pack component data in each line of the image element data. Possible values are 0 (packed 32-bit words) and 1 (filled 32-bit words).

Encoding indicates whether the image element data is run-length encoded or not. Values are 0 (not encoded) and 1 (run-length encoded).

DataOffset indicates the offset to the start of the image element data in bytes from the beginning of the file.

EndOfLinePadding specifies the number of bytes added as padding to the end of each scan line. It is common to pad each line out to the nearest 32-bit boundary. The default value is 0, indicating that no end-of-line padding is present.

EndOfImagePadding specifies the number of bytes added as padding to the end of the image data. It is common to pad the element data out to an even 8K block boundary. The default value is 0, indicating that no end-of-image padding is present.

Description is a 32-byte field storing an ASCII string containing free-form text information describing the contents of the element.

Reserved is a 52-byte field used to pad the header out to 640 bytes in length. Future versions of the DPX format may support additional fields in this area.

Orientation Header

The last of the generic headers is the orientation header. Information in this header describes the position of the image on the display and also contains some additional information on the source of the image data.

Ex.1007 / Page 385 of 1158
APPLE INC.

This header is 256 bytes in length and has the following format:

```
typedef struct _GenericOrientationHeader
{
    DWORD  XOffset;             /* X offset */
    DWORD  YOffset;             /* Y offset */
    SINGLE XCenter;             /* X center */
    SINGLE YCenter;             /* Y center */
    DWORD  XOriginalSize;       /* X original size */
    DWORD  YOriginalSize;       /* Y original size */
    char   FileName[100];       /* Source image file name */
    char   TimeDate[24];        /* Source image date and time */
    char   InputName[32];       /* Input device name */
    char   InputSN[32];         /* Input device serial number */
    WORD   Border[4];           /* Border validity (XL, XR, YT, YB) */
    DWORD  AspectRatio[2];      /* Pixel aspect ratio (H:V) */
    BYTE   Reserved[28];        /* Reserved field used for padding */
} GENERICORIENTATIONHEADER;
```

XOffset and YOffset indicate the offset of the first pixel in the stored image from the first pixel in the original image. If the two images are not offset, then the values of these fields will be 0.

XCenter and YCenter are floating-point values indicating the X and Y coordinates of the center of the image.

XOriginalSize and YOriginalSize indicate the width and height, respectively, of the original image, in pixels per line and lines per image.

FileName is the name of the source image from which the image was processed or extracted.

TimeDate is a 24-byte field containing a time and date stamp of when the source image was created. This stamp is a string in the form *YYYY:MM:DD:HH:MM:SS:LTZ*.

InputName is a 32-byte ASCII string declaring the name of the input device.

InputSN is a 32-byte ASCII string declaring the serial number of the input device.

Border is a set of four values that describe the region of the image eroded due to edge-sensitive filter processing. These values are, in order: X left, X right, Y top, and Y bottom. Values of 0, 0, 0, 0 indicate no border and therefore no erosion.

AspectRatio is the pixel aspect ratio described as a horizontal value (AspectRatio[0]) divided by a vertical value (AspectRatio[1]).

Ex.1007 / Page 386 of 1158
APPLE INC.

Reserved is a 28-byte field used to pad the header out to 256 bytes in length. Future versions of the DPX format may support additional fields in this area.

Industry-Specific Headers

Following the generic headers are two headers that contain industry-specific information. Both of these headers are always present in every DPX file and are initialized with undefined field values when not used. These headers do not contain any information required by the DPX standard and their use is optional.

Film Information Header

The first industry header contains information used by the motion picture film industry. The information in this header describes the film and camera source from which the image frame data was derived. This header is 256 bytes in size and has the following format:

```
typedef struct _IndustryFilmInfoHeader
{
    char   FilmMfgId[2];      /* Film manufacturer ID code */
    char   FilmType[2];       /* File type */
    char   Offset[2];         /* Offset in perfs */
    char   Prefix[6];         /* Prefix */
    char   Count[4];          /* Count */
    char   Format[32];        /* Format */
    DWORD  FramePosition;     /* Frame position in sequence */
    DWORD  SequenceLen;       /* Sequence length in frames */
    DWORD  HeldCount;         /* Held count */
    SINGLE FrameRate;         /* Frame rate of original in frames/sec */
    SINGLE ShutterAngle;      /* Shutter angle of camera in degrees */
    char   FrameId[32];       /* Frame identification */
    char   SlateInfo[100];    /* Slate information */
    BYTE   Reserved[56];      /* Reserved field used for padding */
} INDUSTRYFILMINFOHEADER;
```

FilmMfgId stores the film manufacturer's ID code represented by the same two digits from the film edge code.

FilmType stores the film type code represented by the same two digits from the film edge code.

Offset stores the offset in perfs represented by the same two digits from the film edge code.

Prefix stores the prefix code represented by the same six digits from the film edge code.

Ex.1007 / Page 387 of 1158
APPLE INC.

Count stores the count code represented by the same four digits from the film edge code.

Format is a 32-byte ASCII string specifying the film format.

FramePosition specifies the frame number in the image sequence.

SequenceLen specifies the total number of frames in the image sequence.

HeldCount specifies the number of sequential frames to hold the current frame. This value is used to inject a run of identical frames into a motion picture sequence.

FrameRate stores the frame rate of the original film source in frames per second.

ShutterAngle specifies the shutter (temporal sampling aperture) angle of the motion picture camera in degrees.

FrameId is a 32-byte string identifying the type of frame (key frame, wedge frame, and so on). The format of the data in this field is defined by the creator of the DPX file.

SlateInfo is a 100-byte ASCII string that is used to store production information found on the camera slate.

Reserved is a 56-byte field used to pad the header out to 256 bytes in length. Future versions of the DPX format may support additional fields in this area.

Television Information Header

The second industry header contains information used by the television broadcast industry. The information in this header describes the video signal and television broadcast source from which the image data was derived. This header is 128 bytes in size and has the following format:

```
typedef struct _IndustryTelevisionInfoHeader
{
    DWORD  TimeCode;            /* SMPTE time code */
    DWORD  UserBits;            /* SMPTE user bits */
    BYTE   Interlace;           /* Interlace */
    BYTE   FieldNumber;         /* Field number */
    BYTE   VideoSignal;         /* Video signal standard */
    BYTE   Padding;             /* Structure alignment padding */
    SINGLE HorzSampleRate;      /* Horizontal sampling rate in Hz */
    SINGLE VertSampleRate;      /* Vertical sampling rate in Hz */
    SINGLE FrameRate;           /* Temporal sampling rate or frame rate
                                   in Hz */
    SINGLE TimeOffset;          /* Time offset from sync to first pixel */
```


Ex.1007 / Page 388 of 1158
APPLE INC.

```
    SINGLE Gamma;                /* Gamma value */
    SINGLE BlackLevel;           /* Black level code value */
    SINGLE BlackGain;            /* Black gain */
    SINGLE Breakpoint;           /* Breakpoint */
    SINGLE WhiteLevel;           /* Reference white level code value */
    SINGLE IntegrationTimes;     /* Integration time(s) */
    BYTE   Reserved[76];         /* Reserved field used for padding */
} INDUSTRYTELEVISIONINFOHEADER;
```

TimeCode stores the SMPTE time code value for the image data.

UserBits stores the SMPTE user bits value for the image data.

Interlace is 0 if the image data is not interlaced and 1 if the data is stored using a 2:1 interlace.

FieldNumber indicates the field number to which the image data corresponds, if appropriate. Values are typically 1 or 2 for composite video, 1 to 4 for NTSC, or 1 to 12 for PAL. This value is 0 if the image data is not associated with a particular field.

VideoSignal specifies the video signal standard of the video source. There are 256 possible values to this field; the following are defined:

0	Undefined
1	NTSC
2	PAL
3	PAL-M
4	SECAM
50	YCBCR CCIR 601-2 525-line, 2:1 interlace, 4:3 aspect ratio
51	YCBCR CCIR 601-2 625-line, 2:1 interlace, 4:3 aspect ratio
100	YCBCR CCIR 601-2 525-line, 2:1 interlace, 16:9 aspect ratio
101	YCBCR CCIR 601-2 625-line, 2:1 interlace, 16:9 aspect ratio
150	YCBCR 1050-line, 2:1 interlace, 16:9 aspect ratio
151	YCBCR 1125-line, 2:1 interlace, 16:9 aspect ratio
152	YCBCR 1250-line, 2:1 interlace, 16:9 aspect ratio
200	YCBCR 525-line, 1:1 progressive, 16:9 aspect ratio
201	YCBCR 625-line, 1:1 progressive, 16:9 aspect ratio
202	YCBCR 787.5-line, 1:1 progressive, 16:9 aspect ratio

All values not defined are reserved for use in future versions of DPX.

Padding is a one-byte field used only to maintain the alignment of the structure elements on a 4-byte boundary.

Ex.1007 / Page 389 of 1158
APPLE INC.

HorzSampleRate and VertSampleRate specify the horizontal and vertical sampling rate of the video data in samples per second.

FrameRate specifies the temporal sampling rate, or the frame rate, in samples per second.

TimeOffset stores the time offset from sync (the reference edge of the horizontal sync) to the first pixel in microseconds.

Gamma is the gamma correction exponent value for the image. The default gamma value is 2.2 for an NTSC image.

BlackLevel specifies the digital value representing reference black. The default value for this field differs depending upon the video signal standard of the image data source.

BlackGain defines the linear gain applied to signals below the breakpoint threshold value.

Breakpoint defines the signal level threshold above which the gamma law is applied.

WhiteLevel specifies the digital value representing reference white. The default value for this field differs depending upon the video signal standard of the image data source.

IntegrationTimes indicates the temporal sampling aperture of the television camera.

Reserved is a 76-byte field used to pad the header out to 128 bytes in length. Future versions of the DPX format may support additional fields in this area.

Note that if the value of the VideoSignal field is 0 (undefined), the Gamma, BlackLevel, BlackGain, Breakpoint, and WhiteLevel values cannot be assumed; therefore, these fields must be initialized to their correct values.

User-defined Data

The user-defined data following the industry headers is optional and is not present in a DPX file if the value of the UserSize field in the generic file header is FFFFFFFFh. Otherwise, the value of UserSize indicates the total size of the user-defined data in bytes.

User-defined data may include any information for which the DPX format does not provide storage. Such information includes thumbnail images, additional textual documentation, processing logs, verification signature, color maps, and so on.

Ex.1007 / Page 390 of 1158
APPLE INC.

User-defined data may be from 32 to 1,048,608 bytes in length and has the following format:

```
typedef struct _UserDefinedData
{
    char  UserId[32];     /* User-defined identification string */
    BYTE *Data;           /* User-defined data */
} USERDEFINEDDATA;
```

UserId is a string used to identify the data found in the Data field. This field is used by a DPX file reader to verify that it understands the format of the Data field.

Data is the actual user-defined data. The format of the field is not defined by the DPX standard and the data in this field may be up to one megabyte (1,048,576 bytes) in length.

Image Data

The image data follows the headers and any user-defined data. It is recommended that the image data begin on an 8K block boundary, so there will be padding between the last header, or user-defined data, and the start of the image data. The ImageOffset value in the file header is used to locate the start of the image data.

The image data is stored as an array of 32-bit elements made up of four signed or unsigned character values. Because unsigned values are the default for DPX image data, we will represent the image data array here using the following type definitions:

```
typedef unsigned char BYTE4 [4];
typedef struct _ImageDataElement
{
    BYTE4 *Data;
} IMAGEDATAELEMENT;
```

The DPX format stores image data by pixels, separating each pixel into its component values (also called *samples* in other formats) and storing each value in a separate element (also called a *color plane*). Image data components may be stored in up to eight elements total.

For example, an 8-bit gray-scale pixel contains only one component and is therefore stored as one component value only in one element. A 32-bit RGBA pixel contains four components that may be stored across four elements as four 8-bit component values or may be stored in a single element as one 32-bit component value.

Ex.1007 / Page 391 of 1158
APPLE INC.

As you can see, the DPX format is quite flexible in allowing you to store your image data in any way that makes reading and writing the data the most efficient for your computer hardware.

All components must be the same size across all elements. Valid component sizes are 1-, 8-, 10-, 12-, and 16-bit integers and 32- and 64-bit reals (IEEE floating-point). All components must be stored as words using 32-bit boundaries.

One-bit data is stored in a single element with the first (leftmost) pixel in the least significant bit in the byte and stored as 32-bit data. Eight-bit data is stored four components to a word, 16-bit as two components, 32-bit as one component, and 64-bit components require two 32-bit words to store. All of these component sizes fit evenly within 32-bit word boundaries.

Ten- and 12-bit data, however, does not fit evenly within 32-bit boundaries. In these cases, 10- and 12-bit components are either "filled" into the 32-bit words, leaving unused bits in each word, or "packed" into 32-bit words, leaving no unused bits but causing components to be split across the 32-bit word boundaries.

Filling is accomplished by storing as many components as will fit within a 32-bit word and regarding the remaining bits as padding. For example, three 10-bit components will fit in a 32-bit word using bits 0:9, 10:19, and 20:29. The two remaining bits, 30:31, are not used and are set to zero. Two 12-bit components may be stored in a 32-bit word, but each component is stored starting on a 16-word boundary using bits 0:11 and 16:27. Bits 12:15 and 28:31 are not used for data and only provide alignment padding. One-, 8-, 16-, 32-, and 64-bit data fits evenly within 32-bit boundaries and is therefore never filled.

Packing is used to avoid wasted bits by disregarding all byte and word boundaries. In the previous example, we noted that three 10-bit components use the first 30 bits of a word and waste the last two bits. The fourth 10-bit component then begins on the boundary for the next 32-bit word. If these components were packed, the fourth component would begin on bit 30 of the first word, and would cross over the 32-bit boundary to occupy the first eight bits of the second word.

In the packed format, bits are always stored from the least significant to the most significant bits in a byte. When a component wraps around to the next 32-bit word, its remaining bits are placed in the least significant bit of that word. Packing always stops at the end of a scan line, and any remaining bits are set to zero. (See Figure DPX-2.)

Ex.1007 / Page 392 of 1158
APPLE INC.

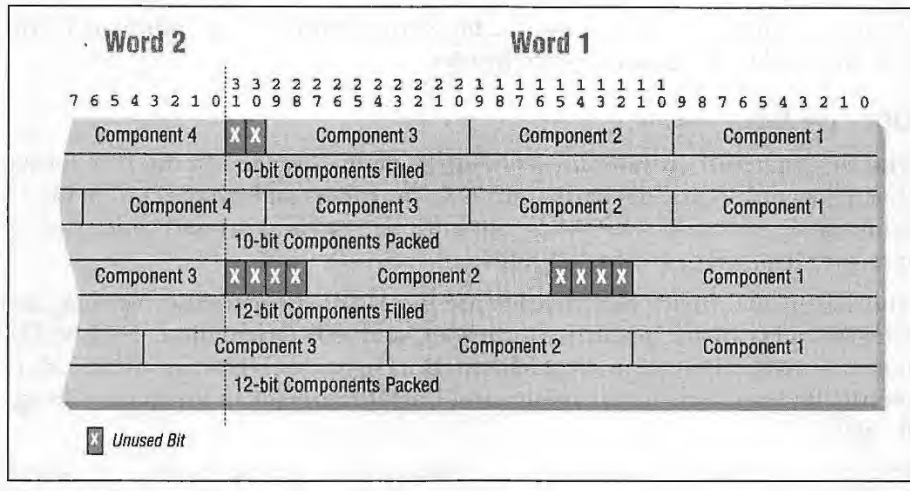


FIGURE DPX-2: *Packed bits*

Lines are typically padded with zero bits to end on 4-byte boundaries, although this is not a requirement of the DPX standard. Image data is also typically padded with zero bits to end on an 8K block boundary.

Data Compression

Components may be stored in an image element using a compressed or uncompressed format. Uncompressed component values are stored directly as raw data without any form of encoding. Optionally, components may be compressed using a simple run-length encoding scheme.

Runs of component values are encoded as a single flag value followed by one or more component values. The least significant bit of the flag value is one if the run of components values are identical, and zero if the run of component values are different. The remaining bits in the flag value specify the number of components in the run. The flag value is always the same size as the component values.

Runs of identical component values are encoded as a flag value followed by a single component value that specifies the value of the run. Runs of different component values are encoded as a flag value followed by a number of components equal to the component count stored in the flag value. A chrominance (CbCr) value is a single component stored as two values.

Ex.1007 / Page 393 of 1158
APPLE INC.

Component run encoding always stops at the end of scan lines. And encoded data is also packed or filled into 32-bit word boundaries, as indicated by the Packing field in the generic image header.

DPX Core Requirements

The DPX standard provides for a minimally defined version of the DPX format that allows quick implementation of DPX file readers and writers, and defines a minimal set of core header fields that must be read and properly initialized to interpret the data stored in a DPX file.

The core header fields of a DPX file are the Magic, ImageOffset, Version, and FileSize fields of the generic file header, and the Orientation, NumberElements, PixelsPerLine, LinesPerElement, DataSign, Descriptor, Transfer, Colorimetric, BitSize, Packing, Encoding, and DataOffset fields of the generic image header.

Minimal DPX file readers must read these fields and may ignore all others. Minimal DPX file writers must initialize these core fields with valid values and may initialize all other header fields with undefined values.

Once all headers, optional user-defined data area, and image data have been defined, the following structure may be used to store the contents of a single DPX file:

```
typedef struct _DpxFileFormat
{
   GENERICFILEHEADER            FileHeader;
   GENERICIMAGEHEADER           ImageHeader;
   GENERICORIENTATIONHEADER     OrientHeader;
   INDUSTRYFILMINFOHEADER       FilmHeader;
   INDUSTRYTELEVISIONINFOHEADER TvHeader;
   USERDEFINEDDATA              *UserData;  /* NULL if data not present */
   IMAGEDATAELEMENT             *ImageData; /* Array of 1 to 8
                                               IMAGEDATAELEMENT structures */
} DPXFILEFORMAT;
```

For Further Information

The DPX file format specification is available in the following document:

ANSI/SMPTE 268M-1994, *SMPTE Standard for File Format for Digital Moving-Picture Exchange (DPX)*, v1.0, 18 February 1994, 14 pages.

Ex.1007 / Page 394 of 1158
APPLE INC.

This document is available directly from SMPTE:

The Society of Motion Picture and Television Engineers
595 W. Hartsdale Avenue
White Plains, NY 10607-1824
Voice: 914-761-1100
FAX: 914-761-3115
Email: *smpte@smpte.org*
WWW: *http://www.smpte.org*

SMPTE is a professional society for motion picture and television engineers that is devoted to advancing the theory and application of motion-imaging technology, including film, video, television, computer imaging, and telecommunications.

The following paper is also a helpful source of information about the DPX format:

Snider, David, Glenn Kennel, Ken Curry, and Michael McCracken, "Digital Moving-Picture Exchange: File Format and Calibration," *SMPTE Journal*, August 1993, pp. 712–714.

The authors are all from the Eastman Kodak Co. in Rochester, New York. Some of them also served on the committee that prepared the SMPTE 268M standard.

Ex.1007 / Page 395 of 1158
APPLE INC.

Dr. Halo

NAME:	Dr. Halo
ALSO KNOWN AS:	CUT, PAL
TYPE:	Bitmap
COLORS:	8-bit maximum
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	64K×64K pixels
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Media Cybernetics
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Dr. Halo
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Used in device independent file interchange

COMMENTS: A well-defined, well-documented format in wide use, which is quick and easy to read and decompress. It lacks, however, a superior compression scheme, making it unsuited for the storage of deep-pixel images.

Overview

The Dr. Halo file format is a device-independent interchange format used for transporting image data from one hardware environment or operating system to another. This format is associated with the HALO Image File Format Library, the Dr. Halo III paint program, and other software applications written and marketed by Media Cybernetics.

Dr. Halo images may contain up to 256 colors, selectable from an 8-bit palette. Only one image may be stored per file. The Dr. Halo format is unusual in that it is divided into two separate files. The first file has the extension .CUT and contains the image data; the second has the extension .PAL and contains the color palette information for the image.

Ex.1007 / Page 396 of 1158
APPLE INC.

File Organization

The Dr. Halo header is shown below:

```
typedef struct _HaloHeader
{
  WORD   Width;          /* 00h   Image Width in Pixels */
  WORD   Height;         /* 02h   Image Height in Scan Lines */
  WORD   Reserved;       /* 04h   Reserved Field (set to 0) */
} HALOHEAD;
```

Width and Height represent the size of the image data.

Reserved is set to zero to allow for possible future expansion of the header.

Following the header is the image data. Each scan line is always encoded using a simple byte-wise run length encoding (RLE) scheme.

File Details

The .CUT file contains image data in the form of a series of scan lines. The first two bytes of each encoded scan line form a Run Count value, indicating the number of bytes in the encoded line. Each encoded run begins with a one-byte Run Count value. The number of pixels in the run is the seven least significant bits of the Run Count byte and ranges in value from 1 to 127. If the most significant bit of the Run Count is 1, then the next byte is the Run Value and should be repeated Run Count times. If the most significant bit is zero, then the next Run Count bytes are read as a literal run. The end of every scan line is marked by a Run Count byte, which may be 00h or 80h.

The following pseudocode illustrates the decoding process:

```
ReadScanLine:
  Read a WORD value of the number of encoded bytes in this scan line
ReadRunCount:
  Read a BYTE value as the Run Count
    If the value of the seven Least Significant Bits (LSB)
      If the Most Significant Bit (MSB)
        Read the next byte as the Run Value and repeat it Run
          Count times
    else
      If the MSB of the Run Count is 0
      Read the next Run Count bytes
    Goto ReadRunCount:
  else
```


Ex.1007 / Page 397 of 1158
APPLE INC.

If the value of the seven LSB of the Run Count is 0
The end of the scan line has been reached
Goto ReadScanLine:

The second Dr. Halo image file usually has the extension .PAL and contains the color palette information for the image. Having a separate color palette file offers the advantage of being able to change the stored colors of an image without re-encoding the image data. The PAL file header is 40 bytes in length and has the following format:

```
typedef struct _HaloPalette
{
BYTE  FileId[2];          /* 00h   File Identifier - always "AH" */
WORD  Version;            /* 02h   File Version */
WORD  Size;               /* 04h   File Size in Bytes minus header */
CHAR  FileType;           /* 06h   Palette File Identifier   */
CHAR  SubType;            /* 07h   Palette File Subtype   */
WORD  BoardId;            /* 08h   Board ID Code */
WORD  GraphicsMode;       /* 0Ah   Graphics Mode of Stored Image   */
WORD  MaxIndex;           /* 0Ch   Maximum Color Palette Index   */
WORD  MaxRed;             /* 0Eh   Maximum Red Palette Value   */
WORD  MaxGreen;           /* 10h   Maximum Green Palette Value   */
WORD  MaxBlue;            /* 12h   Maximum Blue Color Value   */
CHAR  PaletteId[20];      /* 14h   Identifier String "Dr. Halo" */

} HALOPAL;
```

There are actually two types of .PAL files: generic and video hardware-specific. The header shown above is for the generic type. A hardware-specific palette file may contain additional information in the header.

FileId always contains the byte values 41h and 48h.

Version indicates the version of the HALO format to which the palette file conforms.

Size is the total size of the file minus the header. This gives the total size of the palette data in bytes.

FileType, the palette file identifier, is always set to 0Ah.

Subtype, the palette file subtype, is set to 00h for a generic palette file and to 01h for hardware-specific.

BoardId and GraphicsMode indicate the type of hardware and the mode that created and displayed the palette data.

MaxIndex, MaxRed, MaxGreen, and MaxBlue describe the palette data.

Ex.1007 / Page 398 of 1158
APPLE INC.

PaletteId contains up to a 20-byte string with an ASCII identifier. Unused string elements are set to 00h.

Palette data is written as a sequence of three-byte triplets of red, green, and blue values in 512-byte blocks. If a triplet does not fit at the end of a block, the block is padded and the triplet used to start the next block. All RGB values are in the range of 0 to 255.

For Further Information

For further information about the Dr. Halo format, see the specification included on the CD-ROM that accompanies this book. For additional information, contact:

Media Cybernetics
Attn: Bill Shotts
Technical Support Manager
8484 Georgia Avenue
Silver Spring MD 20910
Voice: 301-495-3305, extension 235
FAX: 301-495-5964
WWW: *http://www.mediacy.com/*

Ex.1007 / Page 399 of 1158
APPLE INC.

DVM Movie

NAME:	DVM Movie
ALSO KNOWN AS:	Magic Software DVM
TYPE:	Animation
COLORS:	16 or 256
COMPRESSION:	Pixel packing
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Magic Software
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Magic Software
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes
IMAGES ON CD:	Yes
SEE ALSO:	FLI

USAGE: Storage of small animations and movies.

COMMENTS: A minimal animation format that is of interest mainly because of its simplicity.

Overview

DVM is an animation format created in support of Magic Software. There are several revisions of the format: 1.0, 2.0, 3.0, and 3.1. Appended to the documentation on which this article was based was the following curious statement:

> The DVM format was created by Magic Software and may only be modified by members of Magic Software.

We wish Magic Software well in the enforcement of this policy.

File Organization

DVM files are binary and consist of a header followed by a palette (in v1.0 and higher) and bitmap data organized into a series of frames.

Ex.1007 / Page 400 of 1158
APPLE INC.

File Details

The v1.0 file header consists of the following:

```
typedef struct _DVM_HEADER
{
   char ID[3];        /* File ID "DVM" */
   char Size;         /* Q, F, or V */
   char Info;         /* Flags */
   WORD Wait;         /* Time (ms) to wait between frames */
   WORD Char_count;   /* Number of characters in text, if present */
   char[];            /* Text string, if present */
} DVM_HEADER;
```

For v2.0 and higher, the following header is used:

```
typedef struct _DVM_HEADER
{
   char ID[3];        /* File ID "DVM" */
   char Size;         /* Q, F, or V */
   char Version;      /* High nibble is major version, low nibble is minor */
   char Info;         /* Flags */
   WORD Wait;         /* Time (ms) to wait between frames */
   WORD Char_count;   /* Number of characters in text, if present */
   char[Char_count]; /* Text string, if present */

} DVM_HEADER;
```

The Size field may contain either the characters Q, F, or V, representing Quarter screen, Full screen, and higher Versions, respectively.

The Info field consists of the following flags:

Bit 0 NA
Bit 1 NA
Bit 2 NA
Bit 3 Set if text exists
Bit 4 Set for 256 color (otherwise 16 color)
Bit 5 Set for enhanced palette (otherwise standard palette)
Bit 6 Set for compressed frames
Bit 7 Set for full screen (320×200), otherwise quarter screen (160×100)

Versions 2.0 and higher add support for both 320×200 and 160×100 images. Versions 3.0 and higher add support for both 16- and 256-color images, and versions 3.1 and higher support text. Versions 1.0 or higher should use an enhanced palette.

Ex.1007 / Page 401 of 1158
APPLE INC.

The enhanced palette may contain either 16 or 256 colors. The 16-color palette is organized as 48 consecutive bytes of RGB data. Only the bottom six bits of each byte are significant. The 256-color palette is 768 bytes long and consists of 256 RGB colors. Colors run from 0 to 0xff.

The standard palette is not contained in the file but must be constructed algorithmically by the rendering application. In C pseudocode, this is:

```
typedef struct _RGB { BYTE r,g,b; } RGB;

RGB palette[256];
SHORT i,r,g,b;

for(i = 0;i <= 15;i++)
{
   palette[i].r =
      palette[i].g =
         palette[i].b = (BYTE)(ROUND((DOUBLE)i * 4.2));
}

for(r = 0;r <= 5;r++)
{
   for(g = 0;g <= 5;g++)
      {
      for(b = 0;b <= 5;b++)
         {
         palette[r*36+g+b+16].r = (BYTE)(ROUND((DOUBLE)r * 12.6));
         palette[r*36+g+b+16].g = (BYTE)(ROUND((DOUBLE)g * 12.6));
         palette[r*36+g+b+16].b = (BYTE)(ROUND((DOUBLE)b * 12.6));
         }
      }
}
for(i = 0;i <= 7;i++)
{
   palette[232 + i].r = (BYTE)(i * 9);
   palette[232 + i].g = (BYTE)0;
   palette[232 + i].b = (BYTE)0;

   palette[240 + i].r = (BYTE)0;
   palette[240 + i].g = (BYTE)(i * 9);
   palette[240 + i].b = (BYTE)0;

   palette[248 + i].r = (BYTE)0;
   palette[248 + i].g = (BYTE)0;
   palette[248 + i].b = (BYTE)(i * 9);
}
```

Following the palette is the frame data. The frame origin is in the upper left of the image, and lines are stored in order.

Ex.1007 / Page 402 of 1158
APPLE INC.

Frames may be compressed in a simple version of pixel packing, where bytes of the source data are stored in nibbles of the bytes written to the file. The high-order nibble of the data in the file is extracted first and represents the current byte in the data stream. The low-order nibble of the data byte in the file represents the next byte in the data stream.

For Further Information

For additional information about the DVM movie format, contact:

Bert Greevenbosch
Magic Software Rotterdam
Rotterdam, The Netherlands
Voice: +31-10-4215920
Email: *bert@caiw.nl*

Ex.1007 / Page 403 of 1158
APPLE INC.

Encapsulated PostScript

NAME:	Encapsulated PostScript
ALSO KNOWN AS:	EPS, EPSF, EPSI
TYPE:	Page Description Language (PDL); used as metafile (see article)
COLORS:	Mono
COMPRESSION:	See article
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Adobe
PLATFORM:	Macintosh, MS-DOS, MS Windows, UNIX, others
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Illustration and DTP applications, bitmap and vector data interchange.

COMMENTS: A file format with wide support associated with the PostScript PDL. Although complex, internal language features are well-documented in Adobe's excellent PostScript publications and elsewhere. Many applications, however, write but do not read EPSF-format files, preferring to avoid supporting PostScript rendering to the screen.

Overview

Data in an Encapsulated PostScript (EPSF) file is encoded in a subset of the PostScript Page Description Language (PDL) and then "encapsulated" in the EPS standard format for portable interchange between applications and platforms. An EPSF file is also a special PostScript file that may be included in a larger PostScript language document.

EPSF files are commonly used to contain the graphics and image portions of a document. The main body of the document is defined in one or more PostScript files, but each piece of line art or photographic illustration embedded in the document is stored in a separate EPSF file. This scheme offers

Ex.1007 / Page 404 of 1158
APPLE INC.

several advantages, including the ability to alter illustrations in a document without having to edit the document file itself.

EPSF also provides the ability to store image data in a 7-bit ASCII form, which is occasionally more convenient than the 8-bit binary format used by most bitmap formats.

Although we choose not to discuss PostScript itself in detail, because it is described so extensively elsewhere, we must look briefly at it in order to understand EPSF, which implements a subset of the language.

PostScript was created in 1985 by Adobe Systems and is most often described as a PDL. It is used mainly as a way to describe the layout of text, vector graphics, and bitmap images on a printed or displayed page. Text, color, black-and-white graphics, or photographic-quality images obtained from scanners or video sources can all be described using PostScript. PostScript, however, is a versatile general-purpose computer language, similar in some respects to Forth.

Partly because it is a language, PostScript is device-independent and provides portability and consistent rendering of images across a wide range of platforms. It also implements a de facto industry-standard imaging model for communicating graphics information between applications and hardware devices, such as between word processors and printers. In addition to general-purpose language features, however, PostScript includes commands used for drawing.

A PostScript output device typically contains an interpreter designed to execute PostScript programs. An application sends a stream of PostScript commands (or copies a file to) the device, which then renders the image by interpreting the commands. In the case of printers, typesetters, imagesetters, and film recorders, their main function is to interpret a stream of PostScript language code and render the encoded graphical data onto a permanent medium, such as paper or photographic film. PostScript is capable of handling monochrome, gray-scale, or color images at resolutions ranging from 75 to over 3000 DPI.

PostScript files are written in 7-bit ASCII and can be created using a conventional text editor. Although PostScript can be written by hand, the bulk of the PostScript code produced today comes from applications, which include illustration packages, word processors, and desktop publishing programs. Files produced in this manner are often quite large; it is not unusual to see files in the range of several megabytes.

The PostScript specification is constantly evolving, and two fairly recent developments include Display PostScript and PostScript Level 2. Display PostScript

Ex.1007 / Page 405 of 1158
APPLE INC.

is used for on-screen imaging and is binary-encoded. It is used to drive window-oriented PostScript interpreters for the display of text, graphics, and images. PostScript Level 2 adds additional features to the PostScript Level 1 language, including data compression (including JPEG), device-independent color, improved halftoning and color separation, facsimile compatibility, forms and patterns caching, improved printer support, automatic stroke adjustment, step and repeat capability, and higher operational speeds. PostScript Level 2 code is completely compatible with PostScript Level 1 devices.

File Organization

As we mentioned at the beginning of this section, the EPS format is designed to encapsulate PostScript language code in a portable manner. To accomplish this, an EPS file normally contains nothing but 7-bit ASCII characters, except for the Display EPS format discussed later. An example of a small EPS file is shown in the section called "EPS Files."

File Details

This section contains information about, and examples of, EPS, EPSI, and EPSF files.

EPS Files

An EPS file contains a PostScript header, which is a series of program comments, and may appear as:

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: Figure 1-1, Page 34
%%Creator: The Image Encapsulator
%%CreationDate: 12/03/91 13:48:04
%%BoundingBox:126 259 486 534
%%EndComments
```

Any line beginning with a percent sign (%) in a PostScript file is a comment. Normally, comments are ignored by PostScript interpreters, but comments in the header have special meanings. Encapsulated PostScript files contain two comments that identify the EPS format. The EPS identification comment appears as the first line of the file:

```
%!PS-Adobe-3.0 EPSF-3.0
```

The version number following the "PS-Adobe-" string indicates the level of conformance to the PostScript Document Structuring Conventions. This number

Ex.1007 / Page 406 of 1158
APPLE INC.

will typically be either 2.0 or 3.0. The version number following the "EPSF-" indicates the level of conformance to the Encapsulated PostScript Files Specification and is typically 1.2, 2.0, or 3.0.

The next EPS-specific comment line is:

```
%%BoundingBox:
```

followed by four numeric values, which describe the size and the position of the image on the printed page. The origin point is at the lower-left corner of the page, so a 640×480 image starting at coordinates 0,0 would contain the following comment:

```
%%BoundingBox: 0 0 640 480
```

Both the %%PS-Adobe- and the %%BoundingBox: lines must appear in every EPS file. Ordinary PostScript files may formally be changed into EPS files by adding these two lines to the PostScript header. This, however, is a kludge and does not always work, especially if certain operators are present in the PostScript code (such as initgraphics, initmatrix, initclip, setpageparams, framedevice, copypage, erasepage, and so forth) which are not part of the EPSF subset.

The other comment lines in the header, %%Title:, %%Creator:, and %%CreationDate: are used to identify the name, creator, and creation date of the EPS file. The header is always terminated by the %%EndComments comment. Other comments may also appear in the header, such as %%IncludeFile, %%IncludeFont, and %%Page.

Encapsulated PostScript (EPS) Example

The following shows an example of an EPS file:

```
%!PS-Adobe-2.0 EPSF-1.2
%%Creator: O'Reilly 2.1
%%CreationDate: 12/12/91 14:12:40
%%BoundingBox:126 142 486 651
%%EndComments
/ld {load def} bind def
/s /stroke ld /f /fill ld /m /moveto ld /l /lineto ld /c /curveto ld /rgb
{255 div 3 1 roll 255 div 3 1 roll 255 div 3 1 roll setrgbcolor} def
126 142 translate
360.0000 508.8000 scale
/picstr 19 string def
152 212 1[152 0 0 -212 0 212]{currentfile picstr readhexstring pop}image
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```


Ex.1007 / Page 407 of 1158
APPLE INC.

```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFF8000EE614FFFFFFFFFFFFFFFFFFFFFFFFF
FFFF000000000000000000000000000FFFFFFFF
FFFF0000000000000000000000000000FFFF
FFFF00000000000000000000000000800FFFF
FFFF80000000435C7FFFFFFF600001FFFFFFFF
FFFF0000000000000FFFFFFF700047FFFFFFFF
FFFF00000000000000000000000000001FFFF

Thousands of lines deleted to save space in this book.

FFFF000000000000000000000000000FFFF
FFFF80000000001DF7E61000000001FFFFFFFF
FFFF00000000391FFFFFFFFFE70003FFFFFFFF
FFFF000000000000000000000000001FFFFFFFF
FFFF000000000000000000000000000FFFF
FFFF8000000000000000000000000837DFFFF
FFFF80000000067DFFCF0000000001FFFFFFFFF
FFFF000000000000000000000000FFFFFFFFF
FFFF0000000000000000000000000001FFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
showpage
^D
```

Looking at the EPS example, we can see the comments header at the start of the file. Following the header there is a short block of PostScript code, which does the actual drawing, scaling, cropping, rotating, and so on of the image. This block is sometimes all that needs to be changed in order to alter the appearance of the image.

Following the PostScript code block is bitmap data, which in an EPS file is called a graphics screen representation. This consists of hexadecimal digits. Each byte of image data contains eight pixels of information and is represented by two bytes of hexadecimal data in the bitmap. Image line widths are always a multiple of eight bits and are padded when necessary. Only black-and-white, 1-bit per pixel images are currently supported by EPS.

At the end of the EPS file is the showpage operator, which normally indicates a PostScript output device on which to print or display the completed image or page. In EPS files embedded in other documents, however, showpage is not really needed. Sometimes an EPS file fails to display or import properly because the PostScript interpreter reading the file does not expect to

Ex.1007 / Page 408 of 1158
APPLE INC.

encounter a showpage and becomes confused. You can solve the problem either by disabling the showpage operator in the interpreter or by removing the showpage operator from the EPS files.

If you look at a PostScript file in a editor, you may find that the very last character in the file is a **CTRL-D** (ASCII 04h) character. This control code has a special meaning to a PostScript device; it is an End-Of-Job marker and signals that the PostScript code stream has ended. When a PostScript device reads this character, it may perform an end-of-file terminate-and-reset operation in preparation for the next PostScript data stream. The presence of this character can sometimes be a source of problems to applications that are not expecting or equipped to handle a non-printable ASCII character in the data stream. On the other hand, problems can occur in PostScript output devices if a file does not have this character at the end of the code stream. And if spurious **CTRL-D** characters appear in the data stream, not much will come out of your printer at all.

EPS files (as opposed to normal PostScript) are generally created exclusively by code generators and not by hand. Each line is a maximum of 255 characters wide and is terminated with a carriage return (ASCII 0Dh) on the Macintosh, a linefeed (ASCII 0Ah) under UNIX, and a newline (ASCII 0Dh/0Ah) under MS-DOS. A PostScript interpreter should be able to recognize files using any of these line-termination conventions.

EPS files may also be in preview format. In this format, an actual image file is appended to the end of the file. This provides a quick method of viewing the image contents of the EPS file without having to actually translate the PostScript code. This is handy for applications that cannot handle PostScript interpretation, but which can display bitmap graphics and wish to import EPS files. Previews are typically scaled down (but not cropped), lower-resolution versions of the image. EPS previews are similar to postage stamp images found in the Truevision TGA and Lumena bitmap formats.

Four file formats may be used as EPS preview images: TIFF, Microsoft Windows Metafile (WMF), Macintosh PICT, and EPSI (Encapsulated PostScript Interchange format). In the Macintosh environment, an EPS file is stored only in the file data fork. A PICT preview is stored in the resource fork of the EPS file and will have a resource number of 256. PostScript Level 2 supports JPEG-compressed images as well. TIFF and WMF files are appended to EPS files in their entirety. Because MS-DOS files do not have a resource fork or similar mechanism, a binary header must be prepended to the EPS file containing information about the appended image file.

Ex.1007 / Page 409 of 1158
APPLE INC.

EPS Preview Header

The EPS preview header is 32 bytes in length and has the following format:

```
typedef struct EPSHeader
{
  BYTE Id[4];                   /* Magic Number (always C5D0D3C6h) */
  DWORD PostScriptOffset;       /* Offset of PostScript code */
  DWORD PostScriptLength;       /* Size of PostScrip code */
  DWORD WMFOffset;              /* Offset of WIndows Metafile */
  DWORD WMFSize;                /* Size of Windows Metafile */
  DWORD TIFOffset;              /* Offset of TIFF file */
  DWORD TIFSize;                /* Size of TIFF file */
  DWORD CheckSum;               /* Checksum of previous header fields */
} EPSHEADER;
```

Id, in the first four bytes of the header, contains an identification value. To detect whether an MS-DOS EPS file has a preview section, read the first four bytes of the file. If they are the values C5h D0h D3h C6h, the EPS file has a preview section. Otherwise, the first two bytes of an EPS file will always be 25h 21h (%!).

PostScriptOffset and PostScriptLength point to the start of the PostScript language section of the EPS file. The PostScript code begins immediately after the header, so its offset is always 32.

WMFOffset and WMFSize point to the location of the WMF data if a WMF file is appended for preview; otherwise, these fields are zero.

The same is true for TIFOffset and TIFSize. Because either a TIFF or a WMF file (but not both) can be appended, at least one, and possibly both, sets of the fields will always be zero. If the checksum field is set to zero, ignore it. Offsets are always measured from the beginning of the file.

The three preview formats detailed are only somewhat portable and are fairly device dependent; not all environments can make use of the TIFF and WMF image file formats and fewer still of PICT. For one thing, the addition of 8-bit binary data to the EPS file prevents the file from being transmitted via a 7-bit data path without special encoding. The EPSI format, described in the next section, however, is designed as a completely device-independent method of image data interchange.

Ex.1007 / Page 410 of 1158
APPLE INC.

EPSI Files

EPSI bitmap data is the same for all systems. Its device independence makes its use desirable in certain situations where it is inconvenient to store preview data in 8-bit TIFF, WMF, or PICT format. EPSI image data is encoded entirely in printable 7-bit ASCII characters and requires no uncompression or decoding.

EPSI is similar to EPS bitmap data except that each line of the image begins with a comment % token. In fact, nearly every line in the EPSI preview is a comment; this is to keep a PostScript interpreter from reading the EPSI data as if it were part of the EPS data stored in the file.

Typically, an application will support one or more of the device-dependent preview formats. An application should also support the EPSI format for the export of EPS data to environments that cannot use one of the other preview formats.

The following shows an example of an EPSI file:

```
%%Title: EPSI Sample Image
%%Creator: James D. Murray
%%CreationDate: 12/03/91 13:56:24
%%Pages: 0
%%BoundingBox: 0 0 80 24
%%EndComments
%%BeginPreview: 80 24 1 24
% C0FFFFFFFFFFFFFFFFFF
% 0007F1E18F80FFFFFFFF
% 0007F1C000000001FFFF
% 001FFFE78C01FFFFFFFF
% E7FFFFFFFFFFFFFFFFFF
% 000FFFFF81FFFFFFFFFF
% 0007F38000000001FFFF
% 000FFFCE00000001FFFF
% E1FFFFFFFFC7FFFFFFFF
% 00FFFFFFFFEFFFFFFFFF
% 0003FFCF038008EFFFFF
% 0003FFCF00000000FFFF
% 007FFFFFFFCFFFFFFFFF
% 40FFFFFFFFFFFFFFFFFF
% 0003FFFFFFFBFFFFFFFF
% 00003FFFFFE00001FFFF
% 0001FFFFFFF00031FFFF
% 00FFFFFFFFFFFFFFFFFF
% 0003FFFFFFFFFFFFFFFF
% 00000E000FF80073FFFF
% 00000E000FF80001FFFF
% 0003FFCFFFFFFFFFFFFF
% 0007FFFFFFFFFFFFFFFF
```


Ex.1007 / Page 411 of 1158
APPLE INC.

```
% 000003800071FFFFFFFF
%%EndPreview
%%EndProlog
%%Page: "one: 1
4 4 moveto 72 0 rlineto 0 16 rlineto -72 0 rlineto closepath
8 setlinewidth stroke
%%Trailer
```

EPSF Files

A question that is frequently asked is "What is the difference between an EPS file and an EPSF file?" The answer is that there is no difference; they are the same format. The actual designation, EPSF, is often shortened to EPS, which is also the file extension used for EPSF files on operating systems such as MS-DOS and UNIX.

EPSF files come in two flavors. The first is a plain EPSF file that contains only PostScript code. The second is a Display or Preview EPSF file that has a TIFF, WMF, PICT, or EPSI image file appended to it. Under MS-DOS, Encapsulated PostScript files have the extension .EPS, and Encapsulated PostScript Interchange files have the extension .EPI. On the Macintosh, all PostScript files have the file type EPSF. Also on the Macintosh, a file type of TEXT is allowed for PostScript files created in an editor. Such files should have the extension .EPSF or .EPSI. All other systems should use the filename extensions .EPSF and .EPSI.

For Further Information

PostScript was created and is maintained by Adobe Systems Inc. For specific information about PostScript, contact:

Adobe Systems Inc.
Attn: Adobe Systems Developer Support
1585 Charleston Rd.
P.O. Box 7900
Mountain View, CA 94039-7900
Voice: 415-961-4400
Voice: 800-344-8335
FAX: 415-961-3769
WWW: *http://www.adobe.com/*
FTP: *ftp://ftp.adobe.com/*

Adobe Systems distributes and supports a PostScript Language Software Development Kit to help software developers create applications that use PostScript.

Ex.1007 / Page 412 of 1158
APPLE INC.

The kit includes technical information on PostScript Level 1 and Level 2 compatibility, optimal use of output devices, font software, and file format specifications. Also included are sample fonts, source code, and many PostScript utilities.

There are also numerous books written on PostScript. The fundamental reference set consists of the blue, green, and red books available in bookstores or directly from Adobe Systems or Addison-Wesley:

Adobe Systems, *PostScript Language Tutorial and Cookbook*, Addison-Wesley, Reading, MA, 1985.

Adobe Systems, *PostScript Language Program Design*, Addison-Wesley, Reading, MA, 1988.

Adobe Systems, *PostScript Language Reference Manual*, Second Edition, Addison-Wesley, Reading, MA, 1990.

Other excellent and readily available books about PostScript include:

Braswell, Frank Merritt, *Inside PostScript*, Peachpit Press, Atlanta, GA, 1989.

Glover, Gary, *Running PostScript from MS-DOS*, Wincrest Books, 1989.
Reid, Glenn C., *Thinking in PostScript*. Addison-Wesley, Reading, MA, 1990.

Roth, Stephen F., *Real World PostScript*, Addison-Wesley, Reading, MA, 1988.

Smith, Ross, *Learning PostScript, A Visual Approach*, Peachpit Press, 1990.

A very helpful PostScript resource also exists in the form of a gentleman named Don Lancaster. Mr. Lancaster is the author of more than two dozen books on PostScript, laser printers, and desktop publishing. His articles can be read regularly in *Computer Shopper* magazine. His company, Synergetics, offers many PostScript products and information as well as a free technical support hotline for PostScript users. Contact:

Synergetics
Attn: Don Lancaster
Box 809-PCT
Thatcher, AZ 85552
Voice: 602-428-4073

Ex.1007 / Page 413 of 1158
APPLE INC.

FaceSaver

NAME:	FaceSaver
ALSO KNOWN AS:	None
TYPE:	Bitmap
COLORS:	8-bit
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Metron Computerware, Ltd.
PLATFORM:	UNIX
SUPPORTING APPLICATIONS:	FaceSaver
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: A little-used format, but extremely simple and well-known, in part due to the fact that there is support for this format in the PBM utilities.

COMMENTS: This is an interesting format to examine if you are implementing an ID card system or a similar type of system that deals with the storage of small images (e.g., people's faces).

Overview

The FaceSaver format was created by Lou Katz, and FaceSaver is a registered trademark of Metron Computerware, Ltd. It was created to hold video facial portrait information in a compact, portable, and easy to use form. It is intended to be printed on a PostScript printer. Outside the UNIX graphics world, it is known chiefly because it is supported by the widely used PBM utilities included on the CD-ROM that comes with this book.

The original FaceSaver images were digitized using a Truevision TGA M8 video board.

Ex.1007 / Page 414 of 1158
APPLE INC.

File Organization

Each FaceSaver file consists of several lines of personal information in ASCII format, followed by bitmap data. There must be at least two lines of personal information, and they must include at least the PicData and Image fields.

File Details

The personal information in a FaceSaver file can consist of the following fields:

```
FirstName:
LastName:
E-mail:
Telephone:
Company:
Address1:
Address2:
CityStateZip:
Date:
PicData:    width - height - image bits/pixel
Image:      width - height - bits/pixel
```

Following these fields is a blank line, which is required, and which separates the personal information from the bitmap data that follows it.

The bitmap data consists of ASCII-encoded hexadecimal information, suitable for printing on a PostScript printer. The data is stored in scan-line order, starting from the bottom and continuing to the top of the image, and from left to right. The image data comes originally from a video camera, and is first rotated 90 degrees before being written to a file.

Each pixel is transformed before it's written to the file by multiplying its value by the factor:

```
256 / (max - min)
```

where `max` and `min` are the maximum and minimum pixel values found in the image, respectively.

The Image field in the personal information above is used to correct for non-square pixels. The author writes:

> In most cases, there are 108 (non-square) pixels across in the data, but they would have been 96 pixels across if they were square. Therefore, Image says 96; PicData says 108.

Ex.1007 / Page 415 of 1158
APPLE INC.

For Further Information

For further information about FaceSaver, see the file format specification included on the CD-ROM that accompanies this book. You may also contact the author directly for additional information.

Lou Katz
Email: *lou@orange.metron.com*

Ex.1007 / Page 416 of 1158
APPLE INC.

NAME:	FAX Formats
ALSO KNOWN AS:	FAX, Facsimile File Formats
TYPE:	Bitmap
COLORS:	Mono
COMPRESSION:	RLE, CCITT Group 3, CCITT Group 4
MAXIMUM IMAGE SIZE:	1728×2200 pixels (typical)
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Many
PLATFORM:	All
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	TIFF, PCX, Chapter 9, *Data Compression*

USAGE: Storage of FAX images received through computer-based FAX and FAX-modem boards.

COMMENTS: There is no one single FAX format. The closest to standard are the formats based on a proprietary specification, such as PCX or TIFF. Consider yourself blessed if the format you need to support is one of these.

Overview

There are many facsimile (FAX) file formats, almost as many as there are FAX add-in boards. The PC-based HiJaak Graphics File Conversion Utility (by Inset Systems), as of version 2.1, supports no fewer than 22 different FAX file formats. Each format, however, is basically the same, and consists of a binary header, followed by one or more sections of compressed image data. The data encoding is usually a variant of RLE, CCITT Group 3 or CCITT Group 4. Several FAX formats, in fact, are proprietary variants of better-known formats, such as TIFF and PCX.

Even though all of these FAX file formats were created to store the same kind of image data obtained from the same type of hardware device (i.e., FAX cards), each one is slightly different from all the others. This is problematic.

Ex.1007 / Page 417 of 1158
APPLE INC.

The evolution of the FAX card industry in some ways recapitulates the early evolution of the computer industry as a whole. Each company perhaps imagined it was working alone, or, if not, would quickly come to dominate the market. As the presence of competition became clear, a mad scramble to ship products ensued, and corners were cut. Because companies making FAX cards are by definition hardware-oriented, you can guess where the corners were cut: software.

As the industry started to mature, companies realized that competition was a fact of life, and tried to differentiate their products. One way to do so was through the promulgation of proprietary "standards," designed to keep the originator company one jump ahead of any competition unlucky enough not to be able to push their own specification. Add an unhealthy glop of NIH ("not invented here") spread liberally over the entire FAX board industry, and you have the present situation.

Recently, there have been signs of true maturity in the FAX card industry, with the emergence of the realization that all companies benefit by standardization, and an effort in this direction has been underway for some time. An extension of the TIFF file format, called TIFF Class F, would add the necessary tag extensions to allow easier storage and manipulation of FAX images stored in TIFF format files. (For further information, see the article on TIFF.) At the time of this writing, only one company, Everex, has adopted the unofficial TIFF Class F as its FAX file format standard (perhaps because a now-dead subsidiary of Everex, Cygnet Technologies, pioneered TIFF Class F).

If you need to need to convert FAX file formats, you will need a very versatile conversion utility, such as HiJaak. If you need to write code for an application that reads and writes one or more FAX file formats, you will ordinarily need to contact the manufacturer of the FAX card and obtain any information they are willing to release. If your FAX format is a common one and worth supporting, you should find that you are able to obtain the specifications you need from the manufacturer.

For Further Information

As mentioned above, the best source of information on FAX file formats is from the manufacturer of the FAX card you wish to support. Some FAX card companies publish developers' toolkits for designing software to work with their FAX cards. Unless a company considers its format proprietary, it will have some sort of specification available for their FAX file format.

Ex.1007 / Page 418 of 1158
APPLE INC.

For more information about TIFF Class F, see the TIFF article. You may also be able to obtain the following document:

> Campbell, Joe. *The Spirit of TIFF Class F*, Cygnet Technologies, Berkeley, CA, April 1990.

Cygnet is no longer in business, and Aldus took over support of the TIFF Class F specification. Aldus has recently merged with Adobe Systems, which now holds the copyright to the TIFF specification and administers and maintains the TIFF format.

All information on the TIFF format may now be obtained through the Adobe Developer Support group. However, this group supplies only general TIFF information and does not provide any TIFF tutoring, sample TIFF source code, or sample TIFF files. Contact the Adobe Developer Support group, at *devsupperson@adobe.com*

Questions about the the Adobe Developer's Association should be directed to:

> Adobe Developer's Association
> 1585 Charleston Road
> P.O. Box 7900
> Mountain View CA 94039-7900
> Voice: 415-961-4111
> FAX: 415-967-9231
> FTP: *ftp://ftp.adobe.com/*
> WWW: *http://www.adobe.com/Support/ADA.html*
> BBS: 206-623-6984

Ex.1007 / Page 419 of 1158
APPLE INC.

FITS

NAME:	FITS
ALSO KNOWN AS:	Flexible Image Transport System, FTI
TYPE:	General data format
COLORS:	Unlimited gray scale
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	Two's complement/big-endian
ORIGINATOR:	NOST
PLATFORM:	All
SUPPORTING APPLICATIONS:	FITSview, IMDISP, xv, pbmplus, SAOImage
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes (in FITSview)
IMAGES ON CD:	Yes
SEE ALSO:	VICAR2, PDS

USAGE: FITS is a general-purpose data storage format used primarily for the interchange of data between hardware platforms and software applications.

COMMENTS: FITS is the standard image data storage format for many astronomical organizations, including the astrophysics branch of NASA.

Overview

The FITS image file format is used primarily as a method of exchanging bitmap data between different hardware platforms and software applications that do not support a common image file format. FITS is used mostly by scientific organizations and government agencies that require the storage of astronomical image data (e.g., image data returned by orbital satellites, manned spacecraft, and planetary probes) and ground-based image data (e.g., data obtained from CCD imagery, radio astronomy, and digitized photographic plates).

Although the I in FITS stands for Image, FITS is actually a general-purpose data interchange format. In fact, there is nothing in the FITS specification that limits its use to bitmapped image data.

Ex.1007 / Page 420 of 1158
APPLE INC.

FITS was originally designed explicitly to facilitate the interchange of data between different hardware platforms, rather than between software applications. Much of the FITS data in existence today is (and traditionally, always has been) ground-based and most, if not all, of the agencies and organizations requiring the use of FITS are astronomical in nature.

FITS was originally created by Eric Greisen, Don Wells, Ron Harten, and P. Grosbol and described in a series of papers published in the journal *Astronomy & Astrophysics Supplement.* The NASA/OSSA Office of Standards and Technology (NOST) codified FITS by consolidating these papers into a draft standard of a format for the interchange of astronomical data between scientific organizations. Many such organizations use proprietary imaging software and image file formats not supported by other organizations. FITS, along with VICAR2 and PDS, became a standard interchange format that allows the successful exchange of image data.

FITS is supported by all astronomical image processing facilities and astrophysics data archives. Much of the solar, lunar, and planetary data that is retrieved by the Astrophysics branch of the National Aeronautics and Space Administration (NASA) is distributed using the FITS file format. FITS is currently maintained by a Working Group of the International Astronomical Union (IAU).

Image data normally is converted to FITS not to be stored, but to be imported into another image processing system. Astronomical image data is generally stored in another format, such as the VICAR2 (Video Image Communication and Retrieval) format used by the Multi-Mission Image Processing Laboratory (MIPL).

FITS itself is a very general format capable of storing many types of data, including bitmaps, ASCII text, multidimensional matrices, and binary tables. The simplest FITS file contains a header and a single data stream, called a multidimensional binary array. This is the type of FITS image file we will be examining in this article.

File Organization

In FITS terminology, a basic FITS file contains a primary header and single primary data array. This data structure is known collectively as a Header and Data Unit (HDU). An HDU may contain a header followed by data records, or may contain only a header. All data in a FITS file is organized into logical records 2880 bytes in length.

Ex.1007 / Page 421 of 1158
APPLE INC.

Basically, a FITS file is a header, normally followed by a data stream. Every FITS file begins with an ASCII header which contains one or more logical records. Each logical record is 2880 bytes in size. The last logical record in the header must be padded to 2880 bytes with spaces (ASCII 32).

Each logical record contains 36 records, called card images. A card image is a logical field similar to a data field in a binary image file header. Each card image contains 80 bytes of ASCII data, which describes some aspect of the organization of the FITS image file data. Card images are padded with spaces when necessary to fill out the 80 bytes and do not have delimiters. Card images that are not needed for the storage of a particular set of data contain only spaces.

Most card images may appear in any order within the header, with a few exceptions. The SIMPLE card image must always be first, followed by BITPIX second, NAXIS third, and END last.

Every card image has the following syntax:

```
keyword = value /comment
```

keyword is a 1- to 8-character, left-justified ASCII string that specifies the format and use of the data stored in the card image. If a keyword contains fewer than eight characters, it is padded with spaces. A keyword always occupies columns one through eight in a card image. Only uppercase alphanumerics, hyphens (-), and underscores (_) may be used as characters in a keyword. No lowercase characters, other punctuation, or control codes may be used. If a card image does not have a keyword (the keyword is all spaces), then the card image is treated as a comment.

If the keyword has an associated value, it is then followed by a two-character value indicator (=). This indicator always occupies columns nine and ten in the card image, and if it is present, a value follows the keyword. If the keyword does not have an associated value, then any other ASCII characters may appear in place of the value indicator.

value is an ASCII representation of the numerical or string data associated with the keyword. The value is an ASCII representation of boolean, string, integer, or real (floating-point) data.

comment is an optional field which may follow any value within a card image. A comment is separated from the value by a slash (/) or a space and a slash (/); the latter is recommended. A comment may contain any ASCII characters.

The data in a card image is stored in specific columns. For example, the keyword identifier always occupies columns one through seven in a card image.

Ex.1007 / Page 422 of 1158
APPLE INC.

The keyword value indicator, if present, always occupies columns eight and nine. A boolean value always occupies column 30. And a complex-integer value always occupies columns 31 through 50. Columns that do not contain data are filled with spaces.

Character strings are contained within single quotes. If a string contains a single quote, then it is represented by two consecutive single quotes ('O'Reilly' becomes 'O"Reilly'). All strings contain only 7-bit ASCII values and must be at least eight characters in length, padded with spaces, if necessary. Strings may contain a maximum of 68 characters. Strings may begin with leading spaces, but trailing spaces are considered padding.

All boolean, integer, and floating-point values are represented by their ASCII string equivalents. Boolean variables are represented by the value T or F and are always found in column 30 of the card image. Integer and floating-point values are located in columns 11 through 30 and are right-justified with spaces, if necessary. Complex integers and complex floating-point values are located in columns 31 through 50 and are also right-justified when necessary. All letters used in exponential forms are uppercase.

Examples of valid values are shown below.

Name	Size in Bits	Example
ASCII character	8	'Saturn'
Integer		
Unsigned, one byte	8	127
Unsigned, two bytes	16	32767
Unsigned, four bytes	32	1451342181
Single-precision real		
Fixed-point notation	32	3.14159
Exponential notation	32	0.314159E+01
Double-precision real		
Exponential notation	64	0.3141592653525D+01

Ex.1007 / Page 423 of 1158
APPLE INC.

File Details

This section describes FITS headers and image data.

Keywords

There are many keywords that may be included in FITS headers, and it is unlikely that any FITS reader will understand them all (unrecognized keywords are treated as comments by FITS readers). There are five keywords that are required in every FITS file: SIMPLE, BITPIX, NAXIS, NAXIS*n*, and END. (EXTEND is also a required keyword if extensions are present in the file.) These mandatory keywords are described below:

SIMPLE
The SIMPLE keyword always appears first in any FITS header. The value of this keyword is a boolean value indicating the conformance level of the file to the FITS specification. If the file conforms to the FITS standard, this value is T. If the value is F, then the file differs in some way from the requirements specified in the FITS standard.

BITPIX
The BITPIX card image contains an integer value which specifies the number of bits used to represent each data value. For image data, this is the number of bits per pixel.

BITPIX Value	Data
8	Character or unsigned binary integer
16	16-bit two's complement binary integer
32	32-bit two's complement binary integer
-32	32-bit floating point, single precision
-64	64-bit floating point, double precision

NAXIS
The NAXIS card image contains an integer value in the range of 0 to 999, indicating the number of axes in the data array. Conventional bitmaps have an NAXIS value of 2. A value of 0 signifies that no data follows the header, although an extension may be present.

Ex.1007 / Page 424 of 1158
APPLE INC.

NAXIS*n*
The NAXIS*n* card image indicates the length of each axis in BITPIX units. No NAXIS*n* card images are present if the NAXIS value is 0. The value field of this indexed keyword contains a non-negative integer, representing the number of positions along axis n of an ordinary data array. The NAXIS*n* card image must be present for all values *n* = 1,..., NAXIS. A value of 0 for any of the NAXIS*n* card images signifies that no data follows the header in the HDU. If NAXIS is equal to 0, there should not be any NAXIS*n* keywords.

EXTEND
The EXTEND card image may be included if there are extensions in the FITS file. If there are no extensions, there are no EXTEND card images.

END
The END keyword indicates the end of the header and is always the last card image in a header. END has no value. The card image contains spaces in columns 9 though 80 and is padded out with spaces so that the length of the header is a multiple of 2880 bytes.

Sample Header

The header of a basic FITS image file might appear as follows (the first two lines are for positional information only and are not included in the FITS file):

```
         1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890123456
SIMPLE  =                    T
BITPIX  =                    8/ 8 bits per pixel
NAXIS   =                    2/ Table is a 2D matrix
NAXIS1  =                  168/ Width of table row in bytes
NAXIS2  =                    5/ Number of rows in table
DATE    = '09/17/93'
ORIGIN  = 'O''Reilly & Associates'/ Publisher
AUTHOR  = 'James D. Murray'/ Creator
REFERENC= 'Graphics File Formats'/ Where referenced
COMMENT = 'Sample FITS header'
END
```

For a description of all other valid FITS header keywords, refer to the FITS specification.

Ex.1007 / Page 425 of 1158
APPLE INC.

FITS Image Data

Immediately following the header is the binary image data. This data is stored in 8-bit bytes and is currently never compressed. At the time of this writing, an extension to the FITS standard has been proposed to the FITS community, so future revisions to the FITS standard may incorporate data compression.

The presence or absence of a primary data array is indicated by the values of either the NAXIS or the NAXIS*n* keyword in the primary header.

Data in a FITS file may be stored as bytes, 16- or 32-bit words, and 32- or 64-bit floating-point values that conform to the ANSI/IEEE-754 standard. Fill (ASCII 00h) is added to the data to pad the data out to end on a 2880-byte boundary.

The number of bits of image data, not including the padding added to the end of the image data, may be calculated from the BITPIX, NAXIS, and NAXIS*n* card image values:

```
NumberOfBits = BITBIX * (NAXIS1 * NAXIS2 *...* NAXIS[NAXIS])
```

For Further Information

The specification for FITS is contained in the NOST document included on the CD-ROM that accompanies this book:

> *Implementation of the Flexible Image Transport System (FITS)*, Draft Implementation Standard NOST 100-0.3b., December 1991.

A tutorial and historical guide to FITS is included in the following document, also on the CD-ROM:

> *A User's Guide for FITS*, January 1993.

Both of these documents are also available from the NASA/OSSA Office of Standards and Technology (NOST) FITS Support Office:

> NASA/OSSA Office of Standards and Technology
> Code 633.2
> Goddard Space Flight Center
> Greenbelt, MD 20771
> Voice: 301-441-4189
> Voice: 301-513-1634
> Email: *nost@nssdca.gsfc.nasa.gov*
> Email: *fits@nssdca.gsfc.nasa.gov*
> WWW: *http://www.gsfc.nasa.gov/astro/fits/fits_home.html*

Ex.1007 / Page 426 of 1158
APPLE INC.

The FITS standard is also described in the following references, known collectively as the "Four FITS Papers:"

Wells, D.C., E.W. Greisen, and R.H. Harten, "FITS: a flexible image transport system," *Astronomy and Astrophysics Supplement Series*, vol. 44, 1981, pp. 363–370.

Greisen, E.W. and R.H. Harten, "An extension of FITS for small arrays of data," *Astronomy and Astrophysics Supplement Series*, vol. 44, 1981, pp. 371–374.

Grosbol, P., R.H. Harten, E.W. Greisen, and D.C. Wells, "Generalized extensions and blocking factors for FITS," *Astronomy and Astrophysics Supplement Series*, vol. 73, 1988, pp. 359–364.

Harten, R.H., P. Grosbol, E.W. Greisen, and D.C. Wells, "The FITS tables extension," *Astronomy and Astrophysics Supplement Series*, vol. 73, 1988, pp. 365–372.

Updated information on FITS, including new software applications, frequently appears on the USENET newsgroups *sci.astro.fits* and *sci.data.formats*. Additional software and information on FITS may also be obtained from the following FTP sites:

ftp://fits.cv.nrao.edu/FITS/
ftp://ames.arc.nasa.gov/pub/SPACE/SOFTWARE/
ftp://hypatia.gsfc.nasa.gov/pub/software/

You will find the *sci.data* FAQ at:

http://fits.cv.nrao.edu/traffic/scidataformats/faq.html#fits

You can find FITS images at:

http://astrosun.tn.cornell.edu/FITS.html

FITS is also one of the primary responsibilities of the Working Group on Astronomical Software (WGAS) of the American Astronomical Society. The North American FITS Committee (Dr. Robert J. Hanisch at Space Telescope Science Institute is the chairman) is appointed under the auspices of the WGAS. The WGAS also has a list server, which may be reached by sending a mail message to the following for information on the WGAS mail exploder:

listserv@hypatia.gsfc.nasa.gov

Ex.1007 / Page 427 of 1158
APPLE INC.

Several programs on the CD-ROM display FITS images.

The NSSDC Coordinated Request and Support Office (CRUSO) will provide IMDISP on floppy for a nominal fee. Contact them at:

Voice: 301-286-6695
Email: *request@nssdc.gsfc.nasa.gov*

The FITSIO package contains a collection of subroutines for reading and writing data in the FITS format. This library supports most machines, including Sun, VAX/VMS, Amiga, and the IBM PC and mainframes. It is available via FTP from:

ftp://tetra.gsfc.nasa.gov/pub/fitsio/

Ex.1007 / Page 428 of 1158
APPLE INC.

NAME:	FLI
ALSO KNOWN AS:	FLI Animation, Flic, FLC, FII
TYPE:	Animation
COLORS:	64, 256
COMPRESSION:	Raw, RLE, and delta
MAXIMUM IMAGE SIZE:	320×200, 64K×64K
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Autodesk
PLATFORM:	Intel
SUPPORTING APPLICATIONS:	Autodesk Animator and Animator Pro
SPECIFICATION ON CD:	Yes (summary description by its author)
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	GRASP

USAGE: Used to store animation sequences found in graphics applications, CAD systems, and computer games.

COMMENTS: Currently occupies a market niche being colonized by video.

Overview

The FLI file format (sometimes called Flic) is one of the most popular animation formats found in the MS-DOS and Windows environments today. FLI is used widely in animation programs, computer games, and CAD applications requiring 3D manipulation of vector drawings. Flic, in common with most animation formats, does not support either audio or video data, but instead stores only sequences of still image data.

FLI is popular because of its simple design. It easy to implement FLI readers and writers in software-only applications. FLI also enables quick animation playback and does not require special hardware to encode or decode its data.

FLI is best suited for computer-generated or hand-drawn animation sequences, such as those created using animation and CAD programs. These images achieve the best compression ratios when stored using the FLI format.

Ex.1007 / Page 429 of 1158
APPLE INC.

Natural, real-world images may also be animated by FLI, but such images usually contain a fair amount of noise that will degrade the ability of the FLI encoding algorithms to compress the data and will therefore possibly affect the speed of the animation playback. Also, the fewer the colors in the animation, the better the compression ratio will typically be.

There are two types of FLI animation files. The original FLI format has an extension of .FLI, has a maximum display resolution of 320×200 pixels, and is only capable of supporting 64 colors. This format was created for use by the Autodesk Animator application.

The new FLI format has the extension .FLC, has a maximum display resolution of 64K×64K pixels, and supports up to 256 colors. The data compression scheme used by .FLC files is also more efficient than the scheme used by .FLI files. Applications such as the IBM Multimedia Tool Series, Microsoft Video for Windows, and Autodesk Animator Pro all support .FLC files.

Any application capable of reading the newer .FLC files should be able to read and play back the older .FLI files as well. However, most newer FLI file writers may only have the capability of creating .FLC files. There is really no reason to create .FLI files, unless the animations you are producing must run under software that reads only the .FLI format.

File Organization

FLI animations are sequences of still images called frames. Each frame contains a slice of the animation data. The speed of the animation playback is controlled by specifying the amount of delay that is to occur between each frame.

The data in each frame is always color mapped. Each pixel in a frame contains an index value into a color map defined for that frame. The colors in the map may change from frame to frame as required. And, although the FLI file is limited to displaying a maximum of 256 colors per frame, each pixel is 24 bits in depth, resulting in a palette of more than 16 million colors from which to choose.

The FLI format also supports several types of data compression. Each frame of a FLI animation is typically compressed using an interframe delta encoding scheme. This scheme encodes only the differences between adjacent image frames and not the frames themselves. This strategy results in significantly smaller files than if each frame were independently encoded (intraframe encoding). Interframe encoded data is also very fast to uncompress and display.

Ex.1007 / Page 430 of 1158
APPLE INC.

The first frame of every FLI animation is compressed in its entirety, using a simple run-length encoding algorithm. Because only the differences between each successive frame are encoded, you have to start somewhere. If a frame is delta encoded and the resulting compressed data is larger than the original uncompressed data (quite possible with noisy, natural images), then the frame may be stored uncompressed.

File Details

The header of a FLI file is 128 bytes in length. The first nine fields (22 bytes) are the same for both .FLI and .FLC files. The last ten fields (106 bytes) contain valid data only in .FLC files and are set to 00h in .FLI files.

The FLI file header has the following format:

```
typedef struct _FlicHeader
{
  DWORD FileSize;       /* Total size of file */
  WORD  FileId;         /* File format indicator */
  WORD  NumberOfFrames; /* Total number of frames */
  WORD  Width;          /* Screen width in pixels */
  WORD  Height;         /* Screen height in pixels */
  WORD  PixelDepth;     /* Number of bits per pixel */
  WORD  Flags;          /* Set to 03h */
  DWORD FrameDelay;     /* Time delay between frames */
  WORD  Reserved1;      /* Not used (Set to 00h) */

  // The following fields are set to 00h in a .FLI file

  DWORD DateCreated;    /* Time/Date the file was created */
  DWORD CreatorSN;      /* Serial number of creator program */
  DWORD LastUpdated;    /* Time/Date the file last changed */
  DWORD UpdaterSN;      /* Serial number of updater program */
  WORD  XAspect;        /* X-axis of display aspect ratio */
  WORD  YAspect;        /* Y-axis of display aspect ratio */
  BYTE  Reserved2[38];  /* Not used (Set to 00h) */
  DWORD Frame1Offset;   /* Offset of first frame */
  DWORD Frame2Offset;   /* Offset of second frame */
  BYTE  Reserved3[40];  /* Not used (Set to 00h) */

} FLICHEADER;
```

FileSize contains the total size of the FLI file in bytes.

FileId contains a value identifying the type of Flic file. A value of AF11h indicates an .FLI file, and a value of AF12h indicates an .FLC file.

Ex.1007 / Page 431 of 1158
APPLE INC.

NumberOfFrames contains the total number of frames of animation data. A .FLC file may contain a maximum of 4000 frames; this does not include the ring frame.

Width and Height specify the size of the animation in pixels.

PixelDepth indicates the number of bits per pixel; the value of this field is always 08h.

Flags is always set to 03h, as an indication that the file was properly updated.

FrameDelay indicates the amount of time delay between frames and is used to control the speed of playback. For .FLI files, this value is interpreted in units of 1/70 of a second. For .FLC files, this value is in units of 1/1000 of a second.

Reserved1 is not used and is set to 00h.

DateCreated is an MS-DOS date stamp (the number of seconds occurring since midnight, January 1, 1970) of the date the FLI file was created.

CreatorSN contains the serial number of the Animator Pro application program that created the FLI file. If the file was created by an application using the FlicLib development library, the value of this field will be 46h 4Ch 49h 42h ("FLIB").

LastUpdated is an MS-DOS date stamp indicating the last time the FLI file was modified (also in number of seconds since midnight January 1, 1970).

UpdaterSN contains the serial number of the program that last modified the Flic file.

XAspect and YAspect contain the aspect ratio of the display used to create the animation. For a display with a resolution of 320×200, the aspect ratio is 6:5, and these fields contain the values 6 and 5 respectively. For all other resolutions, the aspect ratio is typically 1:1.

Reserved2 is not used and is set to 00h.

Frame1Offset and Frame2Offset contain the offset of the first and second frames, respectively, of the animation from the beginning of the file. The first frame offset is used to identify the beginning of the animation. The second offset is used as the starting point when the animation loops back to the beginning of the animation from the ring frame.

Reserved3 is not used and is set to 00h.

Ex.1007 / Page 432 of 1158
APPLE INC.

Chunks

All of the data in a FLI file is encapsulated into chunks. Each chunk is a collection of data beginning with a header and followed by the data for that chunk. Chunks may also contains subchunks of data with the same basic format. If a Flic reader encounters a chunk it does not recognize, it should simply skip over it.

Each chunk in a FLI file begins with a 16-byte header that contains the following format:

```
format:
typedef struct _ChunkHeader
{
  DWORD ChunkSize;         /* Total size of chunk */
  WORD  ChunkType;         /* Chunk identifier */
  WORD  NumberOfChunks;    /* Number of subchunks in this chunk */
  BYTE  Reserved[8];       /* Not used (Set to 00h) */

} CHUNKHEADER;
```

ChunkSize is the size of the chunk in bytes. This value includes the size of the header itself and any subchunks contained within the chunk.

ChunkType is an identification value indicating the format of the chunk and the type of data it contains.

NumberOfChunks specifies the number of subchunks contained within this chunk.

Reserved is not used and is set to 00h.

As we have said, a chunk may contain subchunks. In fact, the entire FLI file itself is a single chunk that begins with the FLICHEADER structure. In .FLC files, an optional CHUNKHEADER structure may follow the FLICHEADER structure. This secondary header is called a *prefix header.* If present, this header contains information specific to the Animator Pro application that is not used during the playback of the animation. Other applications can safely skip over the prefix header and ignore the information it contains. Applications other than Animator Pro should never include a prefix header in any .FLC files they create.

For the prefix header, ChunkSize is the size of the entire FLI file minus the 128 bytes of the FLICHEADER. ChunkType is F100h. NumberOfChunks contains the total number of subchunks in the file.

Ex.1007 / Page 433 of 1158
APPLE INC.

Following the prefix header is a series of frame chunks. Each frame chunk contains a single frame of data from the animation. For the frame chunk, ChunkSize is the total number of bytes in the frame, including the header and all subchunks. ChunkType is always F1FAh. NumberOfChunks contains the total number of subchunks in the frame. If the NumberOfChunks value is 0, then this frame is identical to the previous frame, so no color map or frame data is stored, and the previous frame is repeated with the delay specified in the header.

The following lists all the subchunks that may be found in a frame chunk.

ChunkType Value	Chunk Name	Chunk Data Description
04h	COLOR_256	256-level color palette (.FLC files only)
07h	DELTA_FLC	Delta-compressed frame data (.FLC files only)
0Bh	COLOR_64	64-level color palette (.FLI files only)
0Ch	DELTA_FLI	Delta-compressed frame data (.FLI files only)
0Dh	BLACK	Black frame data
0Fh	BYTE_RUN	RLE-compressed frame data
10h	FLI_COPY	Uncompressed frame data
12h	PSTAMP	Postage stamp image (.FLC files only)

The following lists the general internal arrangement of a FLI file:

FLI header
Prefix header (optional)
Frame 1 (RLE compressed)
  PSTAMP subchunk (optional)
  COLOR_256 subchunk (256 colors)
  BYTE_RUN subchunk
  COLOR_256 subchunk (256 colors)
  BYTE_RUN subchunk
Frame 2 (Delta compressed)
  COLOR_256 subchunk (colors different from previous map
  DELTA_FLC subchunk

Ex.1007 / Page 434 of 1158
APPLE INC.

Frame 3 (Uncompressed)
  COLOR_256 subchunk (colors different from previous map)
  FLI_COPY subchunk
Frame 4 (Black)
  BLACK subchunk
Frame *n* (Delta compressed)
  COLOR_256 subchunk (colors different from previous map)
  DELTA_FLC subchunk

Each frame chunk contains at least two subchunks: a color map and the data for the frame. The frame data may be stored in one of several different compressed or uncompressed formats. The first frame of a Flic animation may also contain an additional postage stamp subchunk. Following is an explanation of each subchunk:

DELTA_FLI chunk

The DELTA_FLI chunk contains a single frame of data, which is compressed using delta encoding. The data in this chunk contains the pixel value differences between the current frame and the previous frame. Each scan line of the frame which contains pixel changes is encoded into packets, and only the values of the pixels in the line that have changed are stored.

The DELTA_FLI encoding scheme is an older scheme found mostly in .FLI files, although .FLC files may also contain DELTA_FLI chunks.

The format of a DELTA_FLI chunk is as follows:

```
typedef struct _DeltaFliChunk
{
  CHUNKHEADER Header;    /* Header for this chunk */
  WORD LinesToSkip;      /* Number of initial lines to skip */
  WORD NumberOfLines;    /* Number of encoded lines */

  /* Encoded line (one per 'NumberOfLines') */
  struct _Line
  {
   BYTE NumberOfPackets; /* Number of packets in this line */
   BYTE LineSkipCount;   /* Number of lines to skip */
   struct _Packet        /* Encoded packet (one/NumberOfPackets) */
    {
     BYTE SkipCount;     /* Number of pixels to skip */
     BYTE PacketType;    /* Type of encoding used on this packet */
     BYTE PixelData[];   /* Pixel data for this packet */
    } Packet[NumberOfPackets];
```


Ex.1007 / Page 435 of 1158
APPLE INC.

```
    } Lines[NumberOfLines];

} DELTAFLICHUNK;
```

LinesToSkip contains the number of lines down from the top of the image that are unchanged from the prior frame. This value is used to find the first scan line which contains deltas.

NumberOfLines indicates the number of encoded scan lines in this chunk.

NumberOfPackets indicates the number of packets used to encode this scan line. Each encoded scan line begins with this value.

LineSkipCount is the number of lines to skip to locate the next encoded line.

Each packet in every encoded line contains two values. SkipCount indicates the location of the pixel deltas in this line that are encoded in this packet. PacketType specifies the type of encoding used in this packet. A positive value indicates that the next "PacketType" pixels should be literally read from the chunk and written to the display. A negative value indicates that the absolute value of "PacketType" pixels are to be read literally from the encoded data.

For example, suppose that we have a frame with three encoded scan lines. The first is line number 25, which contains deltas at pixels 4, 22, 23, 24, and 202. The second is line number 97, which contains deltas at pixels 20 and 54 through 67. The third is line number 199, in which all 320 pixels of the line have changed to the same color. The sequence of line and packet field values is shown below:

LinesToSkip	24	Skip 24 lines to first encoded line
NumberOfLines	3	Three encoded lines in this frame
Line		Line 1
NumberOfPackets	3	Three encoded packets in this line
LineSkipCount	71	Skip 71 lines to the next encoded line
Packet		Packet 1
SkipCount	4	Skip 4 pixels
PacketType	1	Read one pixel literally
PixelData	23	New value of pixel 4 is 23
Packet		Packet 2
SkipCount	17	Skip 17 pixels
PacketType	–3	Read one pixel and repeat 3 times
PixelData	65	New value of pixels 22, 23, and 24 is 65
Packet		Packet 3

Ex.1007 / Page 436 of 1158
APPLE INC.

SkipCount	176	Skip 176 pixels
PacketType	1	Read one pixel literally
PixelData	17	New value of pixel 202 is 17
Line		Line 2
NumberOfPackets	2	Two encoded packets in this line
LineSkipCount	102	Skip 102 lines to the next encoded line
Packet		Packet 1
SkipCount	20	Skip 20 pixels
PacketType	1	Read one pixel literally
PixelData	121	New value of pixel 20 is 121
Packet		Packet 2
SkipCount	32	Skip 32 pixels
PacketType	13	Read next 13 pixels literally
PixelData	255	New value of pixels 54 through 67 is 255
Line		Line 3
NumberOfPackets	2	Two encoded packets in this line
LineSkipCount	0	Last encoded line in frame
Packet		Packet 1
SkipCount	0	Start at first pixel in line
PacketType	–256	Read one pixel and repeat 256 times
PixelData	0	New value of pixels 0 though 255 is 0
Packet		Packet 2
SkipCount	256	Skip 256 pixels
PacketType	–64	Read one pixel and repeat 64 times
PixelData	0	New value of pixels 256 though 319 is 0

DELTA_FLC chunk

The DELTA_FLC chunk is a newer version of the DELTA_FLI chunk and is found in all .FLC files. This chunk is essentially the same as the DELTA_FLI chunk with a few field modifications. The PixelData values stored in a DELTA_FLC chunk are 16 bits in size rather than the 8-bit pixel size found in the DELTA_FLI chunk.

The structure of a DELTA_FLC chunk is as follows:

```
typedef struct _DeltaFlcChunk
{
CHUNKHEADER Header;      /* Header for this chunk */
WORD NumberOfLines;      /* Number of encoded lines in chunk */

struct _Line             /* Encoded line (one/'NumberOfLines') */
```


Ex.1007 / Page 437 of 1158
APPLE INC.

```
    {
    WORD PacketCount;      /* Packet count, skip count, or last byte value */
    /*
    ** Additional WORDs of data may appear in this location
    */
    struct _Packet         /* Encoded packet (one per 'Count') */
      {
      BYTE SkipCount;      /* Number of pixels to skip */
      BYTE PacketType;     /* Type of encoding used on this packet */
      WORD PixelData[];    /* Pixel data for this packet */
      } Packet[NumberOfPackets];

    } Lines[NumberOfLines];

} DELTAFLCCHUNK;
```

The number of fields occurring between the PacketCount and the first packet will vary depending upon the value stored in PacketCount. The two most significant bits in PacketCount determine the interpretation of the value stored in this field. If these two bits are 0, then the value is the number of packets occurring in this line. Packet data immediately follows this field and there are no additional WORD values following.

A value of 0 in this field indicates that only the last pixel on the line has changed.

If the most significant bit (bit 15) is 1 and the next bit (bit 14) is 0, the low byte in this WORD is to be stored in the last byte of the current line. A WORD field containing the number of packets in this line follows this value.

If both bits 14 and 15 are set to 1, PacketCount contains a skip count to the next encoded line. PacketCount may then be followed by additional WORD values containing a packet count, skip counts, or last byte values.

BYTE_RUN chunk

When a frame is run-length encoded, the data is stored in a BYTE_RUN chunk. Normally, only the data in the first frame of an animation is encoded using this scheme.

The structure of a BYTE_RUN chunk is as follows:

```
typedef struct _ByteRunChunk
{
  CHUNKHEADER Header;  /* Header for this chunk */
  BYTE PixelData[];    /* RLE pixel data */

} BYTERUNCHUNK;
```


Ex.1007 / Page 438 of 1158
APPLE INC.

Each line in the frame is individually encoded into a series of one or more RLE packets. In the original .FLI format, the first byte of each encoded line was the count of the number of packets used to encode that line, with a packet maximum of 255. The .FLC format, however, allows much longer lines to be used in an animation, and more than 255 packets may be used to encode a line. Therefore, in both .FLC and .FLI files, this initial count byte is read and ignored. Instead, a FLI reader should keep track of the number of pixels decoded to determine when the end of a scan line has been reached.

The RLE scheme used in the BYTE_RUN packet is fairly simple. The first byte in each packet is a type byte that indicates how the packet data is to be interpreted. If the value of this byte is a positive number then the next byte is to be read and repeated "type" times. If the value is negative then it is converted to its absolute value and the next "type" pixels are read literally from the encoded data.

FLI_COPY chunk

This chunk contains a single, uncompressed frame of data. When a frame is stored uncompressed, the FLI_COPY chunk is used. Data is only stored uncompressed when delta or RLE encoding would result in negative compression.

The structure of a FLI_COPY chunk is as follows:

```
typedef struct _CopyChunk
{
  CHUNKHEADER Header;          /* Header for this chunk */
  BYTE PixelData[];            /* Raw pixel data */

} COPYCHUNK;
```

The number of pixels in this chunk is equal to the product of the Width and Height fields (Width*Height) in the FLI file header. FLI_COPY chunks usually result when very complex or noisy images cause the compressed frames to be larger than the uncompressed originals.

PSTAMP chunk

The PSTAMP is a postage stamp of a FLI animation found in the first frame chunk only in .FLC files. This stamp may be a reduced-sized copy of a frame from the animation, possibly from the title screen, that is used as an icon. The size of the stamp is usually 100×63 pixels, but will vary to match the aspect ratio of the frame. This chunk is skipped by FLI readers that do not support the use of PSTAMP chunks.

Ex.1007 / Page 439 of 1158
APPLE INC.

The PSTAMP chunk contains a CHUNKHEADER and two subchunks:

```
typedef struct _PstampChunk
{
  DWORD ChunkSize;              /* Total size of chunk */
  WORD  ChunkType;              /* Chunk identifier */
  WORD  Height;                 /* Height of stamp in pixels */
  WORD  Width;                  /* Width of stamp in pixels */
  WORD  ColorType;              /* Color translation type */
  BYTERUNCHUNK  PixelData;      /* Postage stamp data */

} PSTAMPCHUNK;
```

ChunkSize is the total size of the PSTAMP chunk.

ChunkType value is 0fh, 10h, or 12h.

Height and Width are the height and width of the stamp in pixels.

ColorType indicates the type of color space used by the postage stamp image. This value is always 01h, indicating a six-cube colorspace (see the FLI file format specification for more information on six-cube color space).

Following this header is the postage stamp data chunk.

ChunkType of this header indicates the format of the pixel data. Values are:

0Fh Indicates run-length encoding (a BYTE_RUN chunk)
10h Indicates uncompressed data (a FLI_COPY chunk)
12h Indicates a six-cube color translation table

BLACK chunk

The BLACK chunk represents a single frame of data in which all pixels are set to the color index 0 (normally black) in the color map for this frame. This chunk itself contains no data and has a ChunkType of 0Dh.

The BLACK chunk contains only a CHUNKHEADER:

```
typedef struct _BlackChunk
{
  CHUNKHEADER Header;  /* Header for this chunk */
} BLACKCHUNK;
```


Ex.1007 / Page 440 of 1158
APPLE INC.

COLOR_64 and COLOR_256 chunks

The FLI file format uses a color map to define the colors in an animation. The older .FLI format may have a maximum of 64 colors and stores its color map in a COLOR_64 chunk. A .FLC file may have up to 256 colors and stores its color map in a COLOR_256 chunk. Both of these chunks have the same format:

```
typedef struct _ColormapChunk
{
  CHUNKHEADER Header;          /* Header for this chunk */
  WORD NumberOfElements;       /* Number of color elements in map */
  struct _ColorElement         /* Color element (one per NumberOfElements) */

  {
   BYTE SkipCount;             /* Color index skip count */
   BYTE ColorCount;            /* Number of colors in this element */
   struct _ColorComponent      /* Color component (one /'ColorCount') */
   {
    BYTE Red;                  /* Red component color */
    BYTE Green;                /* Green component color */
    BYTE Blue;                 /* Blue component color */
   } ColorComponents[ColorCount];

  } ColorElements[NumberOfElements];

} COLORMAPCHUNK;
```

The value of ChunkSize in the Header varies depending upon the number of elements in this color map. A chunk containing a color map with 256 elements is 788 bytes in size and therefore ChunkSize contains the value 788.

ChunkType contains a value of 04h for a COLOR_256 chunk or a value of 0Bh for a COLOR_64 chunk.

NumberOfChunks always contains a value of 00h, indicating that this chunk contains no subchunks.

NumberOfElements indicates the number of ColorElement structures in the COLORMAPCHUNK structure. Following this value are the actual ColorElement structures themselves. Each structure contains two fields and one or more ColorComponent structures.

SkipCount indicates the number of color elements to skip when locating the next color map element.

ColorCount indicates the number of ColorComponents structures contained within this ColorElement structure. Following the ColorCount field are the actual ColorComponents structures. Each structure is three bytes in size and contains three fields.

Ex.1007 / Page 441 of 1158
APPLE INC.

The Red, Green, and Blue fields of each ColorComponents structure contain the component values for this color. The range of these field values is 0 to 63 for a COLOR_64 chunk and 0 to 255 for a COLOR_256 chunk.

Normally, an image file contains only one color map. A FLI file, however, allows a color map to be defined for each frame in the animation. Storing a complete color map for each frame would normally require quite a bit of data (768 bytes per frame). FLI files, however, have the capability of storing color maps that contain only the colors that change from frame to frame.

Storing only the deltas in a color map requires that not only the color values be stored, but also their locations in the map. This is accomplished by using a color index value and a skip count. Before a color map value is written, the skip count of the packet is added to the current color index. This sum is the location of the next color map value to write. The number of entries in the packet are written across the same number of entries in the color map. The color index for each color map always starts with the value 0.

For example, the first frame of a .FLC animation always contains a full, 256-element color map. This map is represented by a NumberOfElements value of 1 followed by a single ColorElements structure. This structure will contain a SkipCount value of 0, a ColorCount value of 256, and 256 ColorComponents structures defining the colors in the map. This chunk is 788 bytes in size.

Now, let's say that in the next frame the colors 2, 15, 16, 17, and 197 in the color map are different from those in the first frame. Rather than storing another 788-byte color map chunk, with 251 elements identical to the color map in the previous chunk, we will store only the values and positions of the five color components that changed in the color map.

The color map chunk for the second frame will then contain a NumberOfElements value of 3, followed by three ColorElements structures:

- The first structure will have a SkipCount value of 2, a ColorCount value of 1, and one ColorComponents structure defining the new color values of element 2.
- The second structure will have a SkipCount value of 14, a ColorCount value of 3, and three ColorComponents structures defining the new color values of elements 15, 16, and 17.
- The third structure will have a SkipCount value of 180, a ColorCount value of 1, and one ColorComponents structure defining the new color value of element 197. This chunk will be only 39 bytes in size.

Ex.1007 / Page 442 of 1158
APPLE INC.

The sequence of fields and values for this map is the following:

NumberOfElements	3
ColorElement	
SkipCount	2
ColorCount	1
ColorComponent	R,G,B
ColorElement	
SkipCount	14
ColorCount	3
ColorComponent	R,G,B
ColorComponent	R,G,B
ColorComponent	R,G,B
ColorElement	
SkipCount	180
ColorCount	1
ColorComponent	R,G,B

As you can see, the location of changed color elements is determined by their relative position from the previous changed elements and from their absolute position in the color map. The SkipCount value of the first element is always calculated from the 0th index position. To change the value of element 2, we skip two places, from element 0 to element 2, and change a single component value. To change the values of elements 17, 18, and 19, we make 14 skips from element 2 to element 17 and change the next three component values. We then make 180 skips to element 197 and change the final component value.

Note that if the color map for the current frame is identical to the color map of the previous frame, the color map subchunk need not appear in the current frame chunk.

For Further Information

Autodesk no longer maintains the FLI format and does not formally distribute information about it. However, you may possibly be able to get some information from their homepage:

http://www.autodesk.com

Ex.1007 / Page 443 of 1158
APPLE INC.

For further information about FLI, see the articles by Jim Kent and John Bridges (the author of the format) that are included on the CD-ROM that accompanies this book. In addition, see the following article for information on FLI:

Kent, Jim. "The Flic File Format," *Dr. Dobb's Journal*, March 1993.

Ex.1007 / Page 444 of 1158
APPLE INC.

NAME:	GEM Raster
ALSO KNOWN AS:	IMG
TYPE:	Bitmap
COLORS:	16,384
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	64K×64K
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Digital Research, now part of Novell
PLATFORM:	GEM, MS-DOS, Atari ST
SUPPORTING APPLICATIONS:	DR Paint, DR Doodle, Publishers Paintbrush, and GEM-based applications. Versions of the MS-DOS-based Ventura Publisher were distributed bound to GEM that served mainly to provide GUI services to the application. Many programs on the Atari ST.
SPECIFICATION ON CD:	Yes (for Atari)
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None
USAGE:	Primarily useful in GEM-based application environments.
COMMENTS:	A poorly documented format (in the sense that documentation is hard to come by) in wide use only on the Atari ST platform. It lacks a superior compression scheme and support for included color information. There are at least two versions in existence.

Overview

GEM Raster (also known as IMG) is the native image storage format for the Graphical Environment Manager (GEM), developed and marketed by Digital Research. GEM made its way into the market through OEM bundling deals, special run-time versions bound to products, and as the native operating environment of at least one system, the Atari ST. GEM image files have been important in the PC desktop publishing community due to the bundling deal between

Ex.1007 / Page 445 of 1158
APPLE INC.

Digital Research and the creators of Ventura Publisher, a widely used desktop publishing application.

Although GEM was a contender in the GUI wars some years back, Digital Research's fortunes in this arena declined, and the company was eventually purchased by Novell. Prior to this, however, GEM was distributed by a number of PC hardware manufacturers along with their systems and thus enjoyed a certain currency. GEM raster images may be color, gray scale, or black and white and are always read and written in the big-endian format. Note that several different file formats use the file extension .IMG, a fact that causes confusion in some applications designed to read only GEM raster (IMG) files.

File Organization

Like many other simple bitmap formats, GEM raster files start with a fixed-length header, followed by bitmap data.

File Details

GEM raster files use a 16- or 18-byte header in the following format:

```
typedef struct _GemRaster
{
WORD   Version;          /* Image File Version (Always 1h) */
WORD   HeaderLength;     /* Size of Header in WORDs */
WORD   NumberOfPlanes;   /* Number of Planes */
WORD   PatternLength;    /* Pattern Definition Length */
WORD   PixelWidth;       /* Pixel Width in Micros */
WORD   PixelHeight;      /* Pixel Height in Micros */
WORD   ScanLineWidth;    /* Image Width in Pixels */
WORD   NumberOfLines     /* Image Height in Scan Lines */
WORD   BitImageFlag;     /* Multi-plane GrayColor Flag */

} GEMHEAD;
```

Version always has a value of one.

HeaderLength is either 8 or 9; if the value is 8, there is no BitImageFlag field in the header.

NumberOfPlanes contains the number of bits per pixel of the image source device (a scanner, for instance). This value is typically 1.

PatternLength contains a run-count value, which is usually 1. Any pattern code found in the encoded image data is repeated this number of times.

Ex.1007 / Page 446 of 1158
APPLE INC.

PixelHeight and PixelWidth are the pixel size in microns and are often 85 (55h), corresponding to 1/300 inch, or 300 dpi. The scale of the image may also be determined by using these pixel size values.

ScanLineWidth and NumberOfLines describe the size of the image in lines and pixels.

BitImageFlag indicates whether a multiplane image is color or gray scale. If the BitImageFlag field is present in the header (indicated by a value of 9 in the HeaderLength field) and the image data contains multiple planes (indicated by a value of 2 or greater in the NumberOfPlanes field), a value of 0 indicates color image data and a value of 1 indicates gray-scale image data. If a multiplane image has an 8-field header, then the image is displayed in gray-scale from a fixed, 16-color palette by default. If the image has a 9-field header and only a single plane, the value in the BitImageFlag field is ignored. BitImageFlags was used by GEM-based versions of Ventura Publisher.

The GEM 16-color palette contains the following RGB values:

```
3f, 3f, 3f
3f, 00, 00
00, 3f, 00
3f, 3f, 00
00, 00, 3f
3f, 00, 3f
00, 3f, 3f
2b, 2b, 2b
15, 15, 15
2b, 00, 00
00, 2b, 00
2b, 2b, 00
00, 00, 2b
2b, 00, 2b
00, 2b, 2b
00, 00, 00
```

The GEM 8-bit gray-scale standard palette consists of the following values:

```
ff 7f bf 3f df 5f 9f 1f ef 6f af 2f cf 4f 8f 0f
f7 77 b7 37 d7 57 97 17 e7 67 a7 27 c7 47 87 07
fb 7b bb 3b db 5b 9b 1b eb 6b ab 2b cb 4b 8b 0b
f3 73 b3 33 d3 53 93 13 e3 63 a3 23 c3 43 83 03
fd 7d bd 3d dd 5d 9d 1d ed 6d ad 2d cd 4d 8d 0d
f5 75 b5 35 d5 55 95 15 e5 65 a5 25 c5 45 85 05
f9 79 b9 39 d9 59 99 19 e9 69 a9 29 c9 49 89 09
f1 71 b1 31 d1 51 91 11 e1 61 a1 21 c1 41 81 01
fe 7e be 3e de 5e 9e 1e ee 6e ae 2e ce 4e 8e 0e
f6 76 b6 36 d6 56 96 16 e6 66 a6 26 c6 46 86 06
fa 7a ba 3a da 5a 9a 1a ea 6a aa 2a ca 4a 8a 0a
```


Ex.1007 / Page 447 of 1158
APPLE INC.

```
f2 72 b2 32 d2 52 92 12 e2 62 a2 22 c2 42 82 02
fc 7c bc 3c dc 5c 9c 1c ec 6c ac 2c cc 4c 8c 0c
f4 74 b4 34 d4 54 94 14 e4 64 a4 24 c4 44 84 04
f8 78 b8 38 d8 58 98 18 e8 68 a8 28 c8 48 88 08
f0 70 b0 30 d0 50 90 10 e0 60 a0 20 c0 40 80 00
```

Image data in GEM raster files is always encoded using a simple run-length encoding (RLE) scheme. Data is always encoded and decoded one byte at a time, and there are always eight bits of image data per pixel. For this reason, scan lines are always a multiple of eight pixels in width and are padded when necessary. If the image data contains two or more bits per pixel, then the image will have multiple bit planes.

There are four types of codes in the GEM raster RLE format: vertical replication codes, literal run codes, pattern codes, and encoded run codes. Complicating this RLE scheme is the fact that each of these four codes is a different size, as shown below.

Vertical Replication Code	00 00 FF <Run Count>
Literal Run Code	80 <Run Count 1 to 7F> <'Run Count' Bytes>
Pattern Code	00 <Pattern Length>
Black Run Code	<MSB = 1> <7 LSB = RunCount>
White Run Code	<MSB = 0> <7 LSB = RunCount>

A vertical replication code contains the values 00h 00h FFh, followed by a one-byte count. The count is the number of times to repeat the line that is about to be decoded. A count of one indicates two identical, consecutive lines. A vertical replication code may only appear at the beginning of a scan line. If a replication code is not present at the beginning of a scan line, the line is not repeated.

Literal runs are contiguous lines of pixels that are not encoded. They are written to the encoded data stream as they appear in the bitmap. Literal runs usually appear in encoded image data because data compression had little effect on the pixel data, and it was not efficient to encode the pixels as a run. A literal run code begins with the byte value 80h and is followed by a byte that holds the count value. Following the count are a number of bytes equal to the count value that should be copied literally from the encoded data to the decoded data.

A pattern code begins with the byte 00h and is followed by a byte containing the pattern length. That length is followed by the pattern itself, replicated the number of times specified by the Pattern Length field in the header. Pattern

Ex.1007 / Page 448 of 1158
APPLE INC.

codes are similar to literal run codes, in that the data they contain is not actually compressed in the encoded image data.

Encoded run codes contain only runs of either black or white pixels and are by far the most numerous of all the codes in IMG RLE image data. Black-and-white runs are encoded as a 1-byte packets. Encoded run packets are never 00h or 80h in value. These values are reserved to mark the start of vertical replication codes, pattern codes, and literal run codes. If a byte is read and is not equal to 00h or 80h, the most significant bit indicates the color of the run. If the most significant bit is 1, all the pixels in the run are set to 1 (black). If the most significant bit is 0, the pixels in the run are set to 0 (white). The seven least significant bits in the encoded run are the number of bits in the run. The run may contain 1 to 127 bits.

If an image contains multiple planes, each plane is encoded as the next consecutive scan line of data. One scan line of a four-plane image is encoded as four scan lines of data. The order of the planes is red, green, blue, and intensity value.

The following segment of an encoded scan line:

```
00 00 FF 05  07  8A  02  80 04 2A 14 27 C9  00 03 AB CD EF
```

represents a vertical replication code of five scan lines, a run of seven white bytes (56 pixels), a run of 10 black bytes, (80 pixels), a run of two white bytes (16 pixels), a literal run of four bytes, and pattern code three bytes in length.

For Further Information

The GEM raster format originated at Digital Research, which is now owned by Novell and is currently being supported by DISCUS Distribution Services, a service organization. Note that DISCUS will provide support only if you have first purchased the GEM Programmers's Toolkit from Digital Research. Contact DISCUS at:

DISCUS Distribution Services, Inc.
8020 San Miguel Canyon Road
Salinas, CA 93907-1208
Voice: 408-663-6966

Ex.1007 / Page 449 of 1158
APPLE INC.

You may be able to get some information from Novell/Digital Research at:

Novell/Digital Research, Inc.
P.O. Box DRI
Monterey, CA 93942
Voice: 408-649-3896
Voice: 800-848-1498
BBS: 408-649-3896

We have also been able to include information on Atari support of the GEM raster format on the CD-ROM that accompanies this book.

Ex.1007 / Page 450 of 1158
APPLE INC.

NAME:	GEM VDI
ALSO KNOWN AS:	GEM Vector, VDI, .GDI
TYPE:	Metafile
COLORS:	256
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	32K×32K
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Digital Research
PLATFORM:	GEM GUI running under MS-DOS, some Ataris
SUPPORTING APPLICATIONS:	GEM Artline, GEM Draw Plus, GEM Scan, others
SPECIFICATION ON CD:	Yes (for Atari)
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	GEM Raster

USAGE: Illustration, drawing, and desktop publishing applications, and some data interchange.

COMMENTS: A vector format with good support at one time, associated with the GEM GUI from Digital Research. If you are thinking of supporting this format, be prepared to draw Bezier curves. It also has support for embedded bitmaps.

Overview

Although often called the GEM Vector format, GEM VDI is actually a metafile format and is closely associated with the functioning of the GEM user interface. The GEM system provides a metafile driver that is accessed from within the GEM programming system through a documented API. Display requests to the driver result in items being written to a metafile buffer in GEM's standard metafile format. Metafile elements thus consist of calls to the GEM display system.

Supporting GEM VDI is similar to supporting many other metafile formats. Be prepared to duplicate the functionality of the host system, in this case GEM, or at least a reasonable subset of it, before you're through.

Ex.1007 / Page 451 of 1158
APPLE INC.

File Organization

We would like to have more information on this format. Information provided by DISCUS (see "For Further Information" below) indicates that the file consists of a header followed by a stream of standard-format metafile items.

File Details

The structure for the GEM VDI header is shown below.

```
typedef struct _GemVdiHeader
{
    WORD Identifier;      /* Magic number. Always FFFFh */
    WORD LengthOfHeader   /* Length of the header in 16-bit WORD */
    WORD Version;         /* Format version number */
    WORD Transform;       /* Image origin */
    WORD Coords[4];       /* Size and position of image */
    WORD PageSize[2];     /* Physical page size */
    WORD Bounds[4];       /* Limits of coordinate system */
    WORD Flags;           /* Bit image opcode flag */
} GEMVDIHEADER;
```

Identifier is the magic number of GEM VDI image files. This value is always FFFFh.

LengthOfHeader is the size of the header described as the number of 16-bit WORDs it contains. This value is typically 0Fh.

Version is the version number of the file format. This value is calculated using the formula: 100 * major version number + minor version number.

Transform is the NDC/RC transformation mode flag. This value is 00h if the origin of the image is in the lower-left corner of the display ("Normalized Device Coordinates") and 02h if the origin is in the upper-left corner ("Raster Coordinates").

Coords are four WORDs indicating the minimum and maximum coordinate values of data in the file. These values indicate the size of the image and its position on the display and are stored as: minimum X, minimum Y, maximum X, maximum Y.

PageSize is the size of the physical printed page the image will cover in 1/10 of millimeters. This value is 00h if the page size is undefined by the application creating the image.

Bounds are four WORDs that describe the maximum extent of the coordinate system used by the image and defined by the application. These values are

Ex.1007 / Page 452 of 1158
APPLE INC.

stored as: lower-left X, lower-left Y, upper-right X, upper-right Y.

Flags contains the bit image opcode flag. The values for this field are 00h if no bit image is included in the file and 01h if a bit image is included. Bits 2 through 15 in Flags should always be set to 0.

Standard format metafile items consist of control, integer, and vertex parameters. This structure is described below.

Word	Value	Description
0	control[0]	Opcode
1	control[1]	Vertex count
2	control[3]	Integer parameter count
3	control[5]	Sub-opcode or zero
4	ptsin[0-n]	Input vertex list (if provided)
...		
n+4	intin[0-m]	Input integer (if provided)
...		

Table GEM VDI-1 shows the correspondence of standard metafile items and their opcodes to the display commands accepted by the GEM display subsystem. Arguments appear to be documented only in the GEM Programmer's Toolkit, but you might be able to recover them through diligent application of trial and error.

TABLE GEM VDI-1: *GEM Metafile Items and Commands*

Routine	Code	Subcode	Metafile Object
v_alpha_text	05	19	Output Printer Alpha Text
v_alpha_text	0b	19	Output Printer Alpha Text
v_arc	0b	02	Arc
v_bar	0b	01	Bar
v_bit_image	05	17	Output Bit Image File
v_bit_image	0b	17	Output Bit Image File
v_circle	0b	04	Circle
v_clear_disp_list	05	16	Clear Display List
v_clrwk	03	NA	Clear Workstation
v_ellarc	0b	06	Elliptical Arc
v_ellipse	0b	05	Ellipse
v_ellpie	0b	07	Elliptical Pie Slice

Ex.1007 / Page 453 of 1158
APPLE INC.

GEM VDI (cont'd)

Routine	Code	Subcode	Metafile Object
v_entercur	05	03	Enter Alpha Mode
v_entercur	0b	13	Enter Alpha Mode
v_exitcur	05	02	Exit Alpha Mode
v_fillarea	09	NA	Fill Area
v_form_adv	05	14	Form Advance
v_form_adv	0b	14	Form Advance
v_justified	0b	0a	Justified Graphics Text
v_line	06	NA	Polyline
v_output_window	05	15	Output Window
v_pieslice	0b	03	Pie
v_pieslice	0b	08	Rounded Rectangle
v_pmarker	07	NA	Polymarker
v_qtext	08	NA	Text
v_rfbox	0b	09	Filled Rounded Rectangle
v_updwk	04	NA	Update Workstation
vr_recfl	73	NA	Fill Rectangle
vs_color	0e	NA	Set Color Representation
vsf_clip	81	NA	Set Clipping Rectangle
vsf_color	19	NA	Set Fill Color Index
vsf_interior	17	NA	Set Fill Interior Style
vsf_perimeter	68	NA	Set Fill Perimeter Visibility
vsf_style	18	NA	Set Fill Style Index
vsf_updat	70	NA	Set User Fill
vsl_color	11	NA	Set Polyline Color Index
vsl_ends	6c	NA	Set Polyline End Styles
vsl_type	0f	NA	Set Polyline Type
vsl_udsty	71	NA	Set User Line Style
vsl_width	10	NA	Set Polyline Width
vsm_color	14	NA	Set Polymarker Color Index
vsm_height	13	NA	Set Polymarker Height
vsm_type	12	NA	Set Polymarker Type
vst_alignment	27	NA	Set Graphic Text Alignment
vst_color	16	NA	Set Text Color Index
vst_effects	6a	NA	Set Graphics Text Effects
vst_font	15	NA	Set Text Font
vst_height	0c	NA	Set Character Height
vst_point	6b	NA	Set Character Height (points)
vst_rotation	0d	NA	Set Character Baseline Vector
vswr_mode	20	NA	Set Writing Mode

Ex.1007 / Page 454 of 1158
APPLE INC.

There are also two non-standard metafile items:

v_opnwk	Open Workstation
v_clswk	Close Workstation

There are three metafile escape functions:

v_meta_extents	Update Metafile Extents
v_write_meta	Write Metafile Item
vm_filename	Change GEM VDI Filename

There are several inquire functions:

vq_chcells	Inquire Addressable Character Cells
vq_color	Inquire Color Representation
vq_attributes	Inquire Current Polyline Attributes
vq_extnd	Extended Inquire

Several metafile sub-opcodes are reserved for the Digital Research GEM Output application:

Physical Page Size
Coordinate Window

Several metafile sub-opcodes are also reserved for the Digital Research GEM Draw Plus application:

Group
Set No Line Style
Set Attribute Shadow On
Set Attribute Shadow Off
Start Draw Area Type Primitive
End Draw Area Type Primitive

Also associated with the GEM VDI format is a standard keyboard mapping.

For Further Information

The GEM VDI format originated at Digital Research, which is now owned by Novell, and GEM VDI is currently being supported by DISCUS Distribution Services, a service organization. Note that DISCUS will provide support only if you

Ex.1007 / Page 455 of 1158
APPLE INC.

have first purchased the GEM Programmer's Toolkit from Digital Research. You can contact DISCUS at:

DISCUS Distribution Services, Inc.
8020 San Miguel Canyon Road
Salinas, CA 93907-1208
Voice: 408-663-6966

You may still be able to get some information from Novell/Digital Research at:

Novell/Digital Research, Inc.
P.O. Box DRI
Monterey, CA 93942
Voice: 408-649-3896
Voice: 800-848-1498
BBS: 408-649-3896

We have also been able to include information on Atari support of GEM VDI on the CD-ROM that accompanies this book.

Ex.1007 / Page 456 of 1158
APPLE INC.

NAME:	GIF
ALSO KNOWN AS:	Graphics Interchange Format
TYPE:	Bitmap
COLORS:	1 to 8 bit
COMPRESSION:	LZW
MAXIMUM IMAGE SIZE:	64K×64K pixels
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	CompuServe, Inc.
PLATFORM:	MS-DOS, Macintosh, UNIX, Amiga, others
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes
IMAGES ON CD:	Yes
SEE ALSO:	Chapter 9, *Data Compression*

USAGE: Originally designed to facilitate image transfer and online storage for use by CompuServe and its customers, GIF is primarily an exchange and storage format, although it is based on, and is supported by, many applications.

COMMENTS: A well-defined, well-documented format in wide use, which is quick, easy to read, and reasonably easy to uncompress. It lacks, however, support for the storage of deep-pixel images.

Overview

GIF (Graphics Interchange Format) is a creation of CompuServe and is used to store multiple bitmap images in a single file for exchange between platforms and systems. In terms of number of files in existence, GIF is perhaps the most widely used format for storing multibit graphics and image data. Even a quick peek into the graphics file section of most BBSs and file archives seems to prove this true. Many of these are high-quality images of people, landscapes, cars, astrophotographs, and anthropometric gynoidal data (you guess what that is). Shareware libraries and BBSs are filled with megabytes of GIF images.

Ex.1007 / Page 457 of 1158
APPLE INC.

The vast majority of GIF files contain 16-color or 256-color near-photographic quality images. Gray-scale images, such as those produced by scanners, are also commonly stored using GIF, although monochrome graphics, such as clip art and document images, rarely are.

Although the bulk of GIF files are found in the Intel-based MS-DOS environment, GIF is not associated with any particular software application. GIF also was not created for any particular software application need, although most software applications that read and write graphical image data, such as paint programs, scanner and video software, and most image file display and conversion programs, usually support GIF. GIF was instead intended to allow the easy interchange and viewing of image data stored on local or remote computer systems.

File Organization

GIF is different from many other common bitmap formats in the sense that it is stream-based. It consists of a series of data packets, called *blocks*, along with additional protocol information. Because of this arrangement, GIF files must be read as if they are a continuous stream of data. The various blocks and sub-blocks of data defined by GIF may be found almost anywhere within the file. This uncertainty makes it difficult to encapsulate every possible arrangement of GIF data in the form of C structures.

There are a number of different data block categories, and each of the various defined blocks falls into one of these categories. In GIF terminology, a Graphics Control Extension block is a type of Graphics Control block, for instance. In like manner, Plain Text Extension blocks and the Local Image Descriptor are types of Graphic Rendering blocks. The bitmap data is an Image Data block. Comment Extension and Application Extension blocks are types of Special Purpose blocks.

Blocks, in addition to storing fields of information, can also contain sub-blocks. Each data sub-block begins with a single count byte, which can be in the range of 1 to 255 and indicates the number of data bytes that follow the count byte. Multiple sub-blocks may occur in a contiguous grouping (count byte, data bytes, count byte, data bytes, and so on). A sequence of one or more data sub-blocks is terminated by a count byte with a value of zero.

The GIF format is capable of storing bitmap data with pixel depths of 1 to 8 bits. Images are always stored using the RGB color model and palette data. GIF is also capable of storing multiple images per file, but this capability is rarely

Ex.1007 / Page 458 of 1158
APPLE INC.

utilized, and the vast majority of GIF files contain only a single image. Most GIF file viewers do not, in fact, support the display of multiple image GIF files or may display only the first image stored in the file. For these reasons, we recommend not creating applications that rely on multiple images per file, even though the specification allows this.

The image data stored in a GIF file is always LZW compressed. See Chapter 9 for a discussion of LZW and other compression methods (and also see the sidebar below). This algorithm reduces strings of identical byte values into a single code word and is capable of reducing the size of typical 8-bit pixel data by 40 percent or more. The ability to store uncompressed data, or data encoded using a different compression algorithm, is not supported in the current version of the GIF format.

LZW Is Not Free

If you are creating or modifying software that implements the LZW algorithm, be aware that under certain circumstances, you will need to pay a licensing fee for the use of LZW.

Unisys Corporation owns the patent for the LZW codec (encoding/decoding algorithm) and requires that a licensing fee be paid for each software program that implements the LZW algorithm.

Many people have concluded that the Unisys licensing claim applies only to LZW encoders (software that creates LZW data) and not to LZW decoders (software that only reads LZW data). However, Unisys believes that its patent covers the full LZW codec and requires a licensing fee even for software that reads, but does not write, LZW data.

For more information about the entire issue of LZW licensing, refer to the section called "LZW Legal Issues" in Chapter 9. For a popular alternative to graphics file formats that use LZW, consider using the Portable Network Graphics (PNG) file format, described in Part Two of this book.

There are two revisions of the GIF specification, both of which have been widely distributed. The original revision was GIF87a, and many images were created in this format. The current revision, GIF89a, adds several capabilities, including the ability to store text and graphics data in the same file. If you are

Ex.1007 / Page 459 of 1158
APPLE INC.

supporting GIF, you should include support for both the 87a and 89a revisions. It is a mistake to support only the 89a version, because many applications continue to produce only 87a version files for backward compatibility.

File Details

The "GIF87a" section here discusses features common to both versions; the "GIF89a" section describes only the features added in GIF89a.

GIF87a

Version 87a is the original GIF format introduced in May 1987 and is read by all major software applications supporting the GIF format.

Figure GIF-1 illustrates the basic layout of a GIF87a file. Each file always begins with a Header and a Logical Screen Descriptor. A Global Color Table may optionally appear after the Logical Screen Descriptor. Each of these three sections is always found at the same offset from the start of the file. Each image stored in the file contains a Local Image Descriptor, an optional Local Color Table, and a block of image data. The last field in every GIF file is a Terminator character, which indicates the end of the GIF data stream.

Header

The Header is six bytes in size and is used only to identify the file as type GIF. The Logical Screen Descriptor, which may be separate from the actual file header, may be thought of as a second header. We may therefore store the Logical Screen Descriptor information in the same structure as the Header:

```
typedef struct _GifHeader
{
  // Header
  BYTE Signature[3];      /* Header Signature (always "GIF") */
  BYTE Version[3];        /* GIF format version("87a" or "89a") */
  // Logical Screen Descriptor
  WORD ScreenWidth;       /* Width of Display Screen in Pixels */
  WORD ScreenHeight;      /* Height of Display Screen in Pixels */
  BYTE Packed;            /* Screen and Color Map Information */
  BYTE BackgroundColor;   /* Background Color Index */
  BYTE AspectRatio;       /* Pixel Aspect Ratio */

} GIFHEAD;
```


Ex.1007 / Page 460 of 1158
APPLE INC.

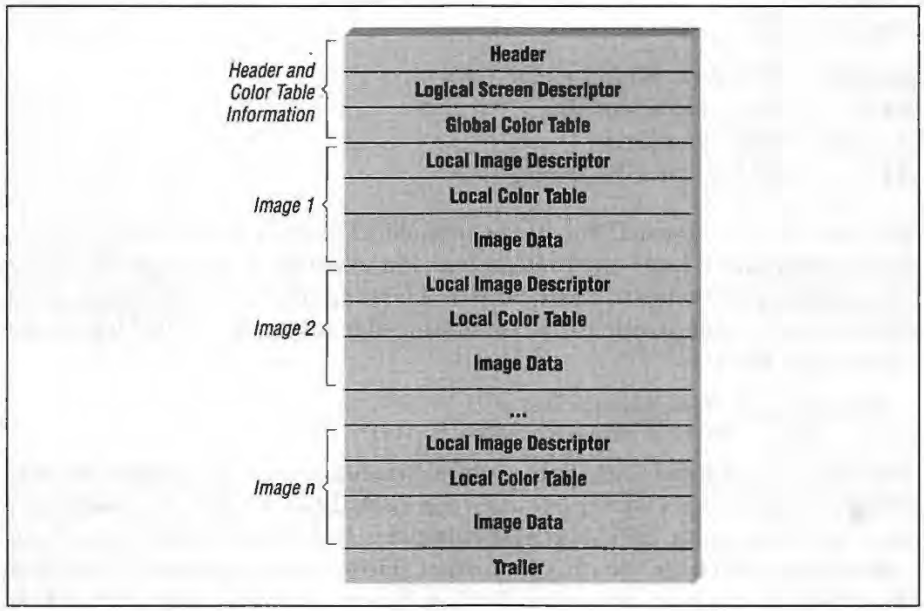


FIGURE GIF-1: *GIF87a file layout*

Signature is three bytes in length and contains the characters GIF as an identifier. All GIF files start with these three bytes, and any file that does not should not be read by an application as a GIF image file.

Version is also three bytes in length and contains the version of the GIF file. There are currently only two versions of GIF: 87a (the original GIF format) and 89a (the new GIF format). Some GIF87a file viewers may be able to read GIF89a files, although the stored image data may not display correctly.

Logical Screen Descriptor

The Logical Screen Descriptor contains information describing the screen and color information used to create and display the GIF file image.

The ScreenHeight and ScreenWidth fields contain the minimum screen resolution required to display the image data. If the display device is not capable of supporting the specified resolution, some sort of scaling will be necessary to properly display the image.

Ex.1007 / Page 461 of 1158
APPLE INC.

Packed contains the following four subfields of data (bit 0 is the least significant bit, or LSB):

Bits 0–2	Size of the Global Color Table
Bit 3	Color Table Sort Flag
Bits 4–6	Color Resolution
Bit 7	Global Color Table Flag

The Size of the Global Color Table subfield contains the number of bits in each Global Color Table entry minus one. For example, if an image contains 8 bits per pixel, the value of this field is 7. The total number of elements in the Global Color Table is calculated by shifting the value one to the left by the value in this field:

```
NumberOfGlobalColorTableEntries =
     (1L << (SizeOfTheGlobalColorTable + 1));
```

The Size of the Global Color Table subfield is always set to the proper size even if there is no Global Color Table (i.e., the Global Color Table Flag subfield is set to 0). If the Color Table Sort Flag subfield is 1, then the Global Color Table entries are sorted from the most important (most frequently occurring color in the image) to the least important. Sorting the colors in the color table aids an application in choosing the colors to use with display hardware that has fewer available colors than the image data. The Sort flag is only valid under version 89a of GIF. Under version 87a, this field is reserved and is always set to 0.

The Color Resolution subfield is set to the number of bits in an entry of the original color palette minus one. This value equates to the maximum size of the original color palette. For example, if an image originally contained eight bits per primary color, the value of this field would be 7. The Global Color Table Flag subfield is set to 1 if a Global Color Table is present in the GIF file, and 0 if one is not. Global Color Table data, if present, always follows the Logical Screen Descriptor header in the GIF file.

BackgroundColor in the Logical Screen Descriptor contains an index value into the Global Color Table of the color to use for the border and background of the image. The background is considered to be the area of the screen not covered by the GIF image. If there is no Global Color Table (i.e., the Global Color Table Flag subfield is set to 0), this field is unused and should be ignored.

Ex.1007 / Page 462 of 1158
APPLE INC.

AspectRatio contains the aspect ratio value of the pixels in the image. The aspect ratio is the width of the pixel divided by the height of the pixel. This value is in the range of 1 to 255 and is used in the following calculation:

```
PixelAspectRatio = (AspectRatio + 15) / 64;
```

If this field is 0, then no aspect ratio is specified.

Global Color Table

The Logical Screen Descriptor may be followed by an optional Global Color Table. This color table, if present, is the color map used to index the pixel color data contained within the image data. If a Global Color Table is not present, each image stored in the GIF file contains a Local Color Table that it uses in place of a Global Color Table. If every image in the GIF file uses its own Local Color Table, then a Global Color Table may not be present in the GIF file. If neither a Global nor a Local Color Table is present, make sure your application supplies a default color table to use. It is suggested that the first entry of a default color table be the color black and the second entry be the color white.

Global Color Table data always follows the Logical Screen Descriptor information and varies in size depending upon the number of entries in the table. The Global Color Table is a series of three-byte triples making up the elements of the color table. Each triple contains the red, green, and blue primary color values of each color table element:

```
typedef struct _GifColorTable
{
  BYTE Red;          /* Red Color Element       */
  BYTE Green;        /* Green Color Element     */
  BYTE Blue;         /* Blue Color Element      */

} GIFCOLORTABLE;
```

The number of entries in the Global Color Table is always a power of two (2, 4, 8, 16, and so on), up to a maximum of 256 entries. The size of the Global Color Table in bytes is calculated by using bits 0, 1, and 2 in the Packed field of the Logical Image Descriptor in the following way:

```
ColorTableSize = 3L * (1L << (SizeOfGlobalColorTable + 1));
```

The Header, Logical Screen Descriptor, and Global Color Map data are followed by one or more sections of image data. Each image in a GIF file is stored

Ex.1007 / Page 463 of 1158
APPLE INC.

separately, with an Image Descriptor and possibly a Local Color Table. The Image Descriptor is similar to a header and contains information only about the image data that immediately follows it. The Local Color Table contains color information specific only to that image data and may or may not be present.

Local Image Descriptor

The Local Image Descriptor appears before each section of image data and has the following structure:

```
typedef struct _GifImageDescriptor
{
  BYTE Separator;     /* Image Descriptor identifier */
  WORD Left;          /* X position of image on the display */
  WORD Top;           /* Y position of image on the display */
  WORD Width;         /* Width of the image in pixels */
  WORD Height;        /* Height of the image in pixels */
  BYTE Packed;        /* Image and Color Table Data Information */
} GIFIMGDESC;
```

Separator contains the value 2Ch and denotes the beginning of the Image Descriptor data block.

Left and Top are the coordinates in pixels of the upper-left corner of the image on the logical screen. The upper-left corner of the screen is considered to be coordinates 0,0.

Width and Height are the size of the image in pixels.

Packed contains the following five subfields of data (bit 0 is the LSB):

Bit 0	Local Color Table Flag
Bit 1	Interlace Flag
Bit 2	Sort Flag
Bits 3–4	Reserved
Bits 5–7	Size of Local Color Table Entry

The Local Color Table Flag subfield is 1 if a Local Color Table is associated with this image. If the value of this subfield is 0, then there is no Local Color Table present, and the Global Color Table data should be used instead.

The Interlace Flag subfield is 1 if the image is interlaced and 0 if it is non-interlaced. (See the description of Image Data for an explanation of interlaced image data.)

Ex.1007 / Page 464 of 1158
APPLE INC.

The Sort Flag subfield indicates whether the entries in the color table have been sorted by their order of importance. Importance is usually decided by the frequency of occurrence of the color in the image data. A value of 1 indicates a sorted color table, while a value of 0 indicates a table with unsorted color values. The Sort Flag subfield value is valid only under version 89a of GIF. Under version 87a, this field is reserved and is always set to 0.

The Size of Local Color Table Entry subfield is the number of bits per entry in the Local Color Table. If the Local Color Table Flag subfield is set to 0, then this subfield is also set to 0.

Local Color Table

If a Local Color Table is present, it immediately follows the Local Image Descriptor and precedes the image data with which it is associated. The format of all Local Color Tables is identical to that of the Global Color Table. Each element is a series of 3-byte triples containing the red, green, and blue primary color values of each element in the Local Color Table:

```
typedef struct _GifColorTable
{
  BYTE Red;             /* Red Color Element */
  BYTE Green;           /* Green Color Element */
  BYTE Blue;            /* Blue Color Element */

} GIFCOLORTABLE;
```

The number of entries and the size in bytes of the Local Color Table is calculated in the same way as the Global Color Table:

```
ColorTableSize = 3L * (1L << (SizeOfLocalColorTable + 1));
ColorTableNumberOfEntries = 1L << (SizeOfLocalColorTable
     + 1);
```

A Local Color Table only affects the image it is associated with and, if it is present, its data supersedes that of the Global Color Table. Each image may have no more than one Local Color Table.

Image data

GIF files do not compress well when stored using file archivers such as pkzip and zoo. This is because the image data found in every GIF file is always compressed using the LZW (Lempel-Ziv-Welch) encoding scheme, the same compression algorithm used by most file archivers. (See the sidebar about LZW at the beginning of this article.) Compressing a GIF file is therefore a redundant operation, which rarely results in smaller files and is usually not worth the time and effort involved in the attempt.

Ex.1007 / Page 465 of 1158
APPLE INC.

Normally when LZW-encoded image data is stored in a graphics file format, it is arranged as a continuous stream of data that is read from beginning to end. The GIF format, however, stores encoded image data as a series of data sub-blocks.

Each data sub-block begins with a count byte. The value of the count byte may range from 1 to 255 and indicates the number of data bytes in the sub-block. The data blocks immediately follow the count byte. A contiguous group of data blocks is terminated by a byte with a zero value. This may be viewed as either a terminator value or as a sub-block with a count byte value of zero; in either case, it indicates that no data bytes follow.

Because GIF files do not contain a contiguous stream of LZW-encoded data, each sub-block must be read and the data sent to an LZW decoder. Most sub-blocks storing image data will be 255 bytes in length, so this is an excellent maximum size to use for the buffer that will hold the encoded image data. Also, the LZW encoding process does not keep track of where each scan line begins and ends. It is therefore likely that one scan line will end and another begin in the middle of a sub-block of image data.

The format of the decoded GIF image data is fairly straightforward. Each pixel in a decoded scan line is always one byte in size and contains an index value into either a Global or Local Color Table. Although the structure of the GIF format is quite capable of storing color information directly in the image data (thus bypassing the need for a color table), the GIF specification does not specify this as a possible option. Therefore, even 1-bit image data must use 8-bit index values and a 2-entry color table.

GIF image data is always stored by scan line and by pixel. GIF does not have the capability to store image data as planes, so when GIF files are displayed using plane-oriented display adapters, quite a bit of buffering, shifting, and masking of image data must first occur before the GIF image can be displayed.

The scan lines making up the GIF bitmap image data are normally stored in consecutive order, starting with the first row and ending with the last. The GIF format also supports an alternate way to store rows of bitmap data in an interlaced order. Interlaced images are stored as alternating rows of bitmap data. If you have ever viewed a GIF file that appeared on the screen as a series of four "wipes" that jumped across the screen as the image was displayed, you were viewing an interlaced GIF file.

Ex.1007 / Page 466 of 1158
APPLE INC.

Figure GIF-2 compares the order of rows stored in an interlaced and non-interlaced format. In the non-interlaced format, the rows of bitmap data are stored starting with the first row and continuing sequentially to the last row. This is the typical storage format for most bitmap file formats. The interlaced format, however, stores the rows out of the normal sequence. All the even rows are stored first and all the odd rows are stored last. We can also see that each successive pass usually encodes more rows than the previous pass.

GIF uses a four-pass interlacing scheme. The first pass starts on row 0 and reads every eighth row of bitmap data. The second pass starts on the fourth row and reads every eighth row of data. The third pass starts on the second row and reads every fourth row. The final pass begins on the first row and reads every second row. Using this scheme, all of the rows of bitmap data are read and stored.

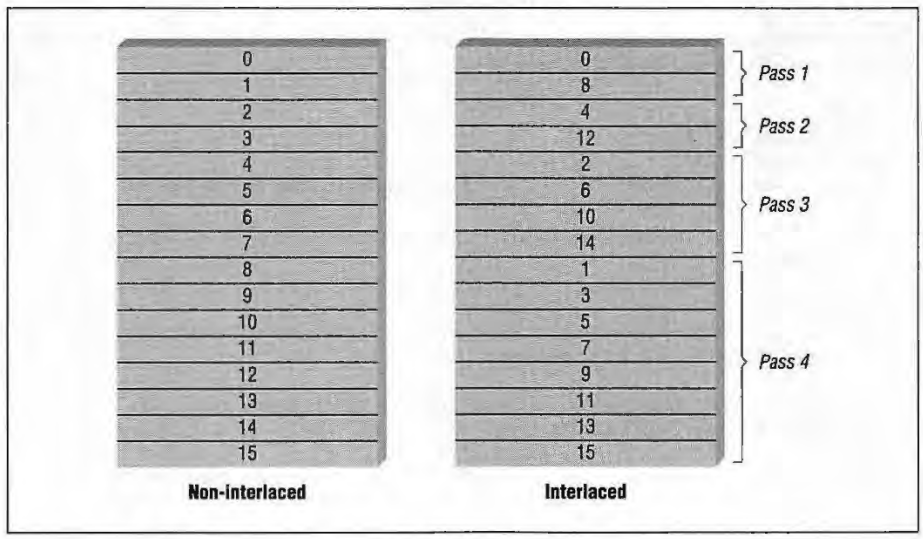


FIGURE GIF-2: *Arrangement of interlaced and non-interlaced scan lines*

Why interlace a GIF image? Interlacing might seem to make the reading, writing, and displaying of the image data more difficult, and of course it does. Does this arrangement somehow make the image easier to display on interlaced monitors? The answer lies in one of the original purposes of GIF.

Ex.1007 / Page 467 of 1158
APPLE INC.

GIF was designed as an image communications protocol used for the interactive viewing of online images. A user connected to an information service via a modem could not only download a GIF image, but could also see it appear on his or her display screen as it was being downloaded. If a GIF image were stored in a non-interlaced format, the GIF image would display in a progressive fashion starting at the top of the screen and ending at the bottom. After 50 percent of the download was completed, only the top half of the GIF image would be visible. An interlaced image, however, would display starting with every eighth row, then every fourth row, then every second row, and so on. When the download of an interlaced GIF image was only 50 percent complete, the entire contents of the image could be discerned even though only half the image had been displayed. The viewer's eye and brain would simply fill in the missing half.

Interlacing presents a problem when converting a GIF image from one format to another. A scan-line table must be created to write out the scan lines in their proper, non-interlaced order. The following sample code is used to produce a scan-line table of an interlaced image:

```
WORD i, j;
WORD RowTable1[16];
WORD RowTable2[16];
WORD ImageHeight = 16;              /* 16 lines in the GIF image */

for (i = 0; i < ImageHeight; i++)     /* Initialize source array*/
     RowTable1[i] = i;

j = 0;
for (i = 0; i < ImageHeight; i += 8, j++)  /* Interlace Pass 1 */
     RowTable2[i] = RowTable1[j];

for (i = 4; i < ImageHeight; i += 8, j++)  /* Interlace Pass 2 */
     RowTable2[i] = RowTable1[j];

for (i = 2; i < ImageHeight; i += 4, j++)  /* Interlace Pass 3 */
     RowTable2[i] = RowTable1[j];

for (i = 1; i < ImageHeight; i += 2, j++)  /* Interlace Pass 4 */
     RowTable2[i] = RowTable1[j];
```

The array RowTable1[] contains the mapping of the scan lines in a non-interlaced image, which in this example are the values 0 to 15 in consecutive order. The array RowTable2[] is then initialized by the interlacing code to contain the mapping of the scan lines of the interlaced image:

Ex.1007 / Page 468 of 1158
APPLE INC.

RowTable1[]	RowTable2[]
0	0
1	8
2	4
3	9
4	2
5	10
6	5
7	11
8	1
9	12
10	6
11	13
12	3
13	14
14	7
15	15

We can restore the non-interlaced image by stepping through the values stored in RowTable2[]. The 0th row of the non-interlaced image is the 0th row of the interlaced image. The first row of the non-interlaced image is the eighth row of the interlaced image. The second row of the non-interlaced image is the fourth row of the interlaced image, and so on.

Trailer

The Trailer is a single byte of data that occurs as the last character in the file. This byte value is always 3Bh and indicates the end of the GIF data stream. A trailer must appear in every GIF file.

GIF89a

Version 89a is the most recent revision of the GIF image file format and was introduced in July of 1989. Although the GIF89a format is very similar to GIF 87a, it contains several additional blocks of information not defined in the 87a specification. For this reason GIF89a image files may not be read and displayed properly by applications that read only GIF87a image files. Many of these programs do not not attempt to display an 89a image file, because the version number "89a" will not be recognized. Although changing the version number from "89a" to "87a" will solve this problem, the GIF image data may still not display properly, for reasons we shall soon see.

Figure GIF-3 illustrates the basic layout of a GIF89a image file. Just as with version 87a, the 89a version also begins with a Header, a Logical Screen

Ex.1007 / Page 469 of 1158
APPLE INC.

Descriptor, and an optional Global Color Table. Each image also contains a Local Image Descriptor, an optional Local Color Table, and a block of image data. The trailer in every GIF89a file contains the same values found in 87a files.

Version 89a added a new feature to the GIF format called Control Extensions. These extensions to the GIF87a format are specialized blocks of information used to control the rendering of the graphical data stored within a GIF image file. The design of GIF87a only allowed the display of images one at a time in a "slide show" fashion. Through the interpretation and use of Control Extension data, GIF89a allows both textual and bitmap-based graphical data to be displayed, overlaid, and deleted as in an animated multimedia presentation.

The four Control Extensions introduced by GIF89a are the Graphics Control Extension, the Plain Text Extension, the Comment Extension, and the Application Extension, summarized here and described in greater detail in the sections below.

Graphics Control Extension blocks control how the bitmap or plain-text data found in a Graphics Rendering block is displayed. Such control information includes whether the graphic is to be overlaid in a transparent or opaque fashion over another graphic, whether the graphic is to be restored or deleted, and whether user input is expected before continuing with the display of the GIF file data.

Plain Text Extension blocks allow the mixing of plain-text ASCII graphics with bitmapped image data. Many GIF images contain human-readable text that is actually part of the bitmap data itself. Using the Plain Text Extension, captions that are not actually part of the bitmapped image may be overlaid onto the image. This can be invaluable when it is necessary to display textual data over an image, but it is inconvenient to alter the bitmap to include this information. It is even possible to construct an 89a file that contains only plain-text data and no bitmap image data at all.

Comment Extension blocks contain human-readable ASCII text embedded in the GIF data stream that is used in a manner similar to program comments in C language code.

Application Extension blocks allow the storage of data that is understood only by the software application reading the GIF file. This data could be additional information used to help display the image data or to coordinate the way the image data is displayed with other GIF image files.

Ex.1007 / Page 470 of 1158
APPLE INC.

With only a few restrictions, any number of Control Extension blocks may appear almost anywhere in a GIF data stream following the Global Color Table. All Extension blocks begin with the Extension Introducer value 21h, which identifies the block of data as an Extension block. This value is followed by a Block Label, which identifies the type of extension information contained within the block. Block Label identification values range from 00h to FFh. The Plain Text, Application, and Comment Extension blocks may also contain one or more sub-blocks of data.

Interestingly enough, all of the Control Extension features added by 89a are optional and are not required to appear in a GIF data stream. The only other difference between 87a and 89a is that at least one of the Image Descriptor and Logical Screen Descriptor fields, which are reserved under 87a, is used under 89a. In fact, any GIF files that are written under version 89a, but do not use any of the 89a features, should use the version number GIF87a.

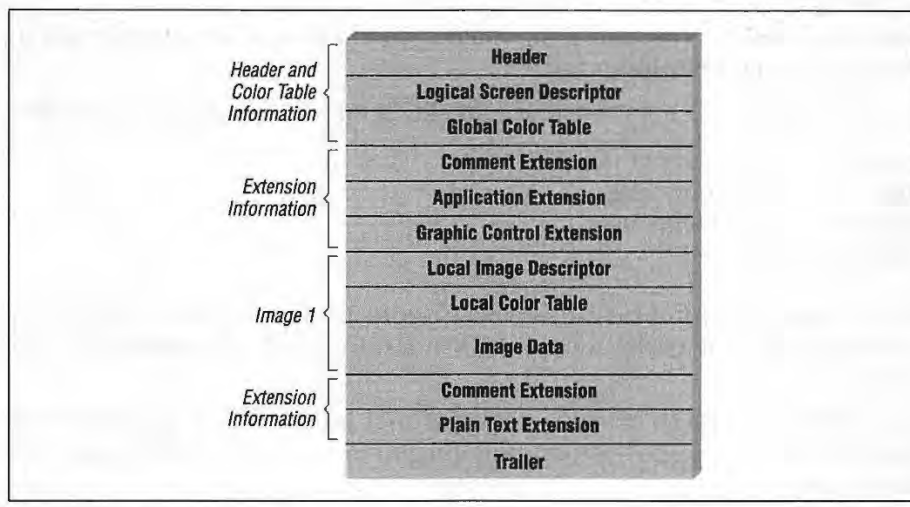


FIGURE GIF-3: *Layout of a GIF89a file*

Graphics Control Extension block

The information found in a Graphics Control block is used to modify the data in the Graphical Rendering block that immediately follows it. A Graphics Control block may modify either bitmap or plain-text data. It must also occur in the GIF stream before the data it modifies, and only one Graphics Control block may appear per Graphics Rendering block.

Ex.1007 / Page 471 of 1158
APPLE INC.

The Graphics Control Extension block is eight bytes in length and has the following structure:

```
typedef struct _GifGraphicsControlExtension
{
  BYTE Introducer;    /* Extension Introducer (always 21h) */
  BYTE Label;         /* Graphic Control Label (always F9h) */
  BYTE BlockSize;     /* Size of remaining fields (always 04h) */
  BYTE Packed;        /* Method of graphics disposal to use */
  WORD DelayTime;     /* Hundredths of seconds to wait       */
  BYTE ColorIndex;    /* Transparent Color Index */
  BYTE Terminator;    /* Block Terminator (always 0) */

} GIFGRAPHICCONTROL;
```

Introducer contains the value 21h and is used to identify the start of a Extension data block.

Label contains the value F9h and is used to identify this block of data as a Graphics Control Extension.

BlockSize contains the value 04h, which is the number of bytes in the Packed, DelayTime, and ColorIndex fields.

Packed contains the following four subfields of information (bit 0 is the LSB):

Bit 0	Transparent Color Flag
Bit 1	User Input Flag
Bits 2–4	Disposal Method
Bits 5–7	Reserved

If the Transparent Color Flag subfield is set to 1, the ColorIndex field of this extension contains a color transparency index. If no index is present, this bit is set to 0.

The User Input Flag subfield is set to 1 if user input (key press, mouse click, and so forth) is expected before continuing to the next graphic sequence; otherwise, this bit is set to zero.

The Disposal Method subfield contains a value indicating how the graphic is to be disposed of once it has been displayed. The currently defined values for this field are 00h (disposal method not specified), 01h (do not dispose of graphic), 02h (overwrite graphic with background color), and 04h (overwrite graphic with previous graphic).

Ex.1007 / Page 472 of 1158
APPLE INC.

The Reserved subfield is not used in GIF89a and is always set to 0.

DelayTime in the Graphics Control Extension block contains a value equal to the number of hundredths of a second that must elapse before the graphics presentation continues. If this field is 0, then no delay is used. If both this delay and the user input bit is set, the graphic continues when either the delay expires or user input is received.

ColorIndex is the color transparency index. This field contains a value only if the Transparent Color Flag subfield in the Packed field is set to 1.

Terminator contains the value 0 and marks the end of the Graphics Control Extension block.

Plain Text Extension block

GIF87a files may contain bitmapped data only in the form of a Graphical Rendering block. GIF89a adds the ability to store textual information that may be rendered as a graphical image.

Any number of Plain Text Extension blocks may appear in a GIF file. To display plain-text data, a grid is described that contains the data. The height, width, and position of the grid on the display screen are specified. The size of each cell in the grid is also described, and one character is displayed per cell. The foreground and background color of the text are taken from the Global Color Table and are also described in the Plain Text Extension block. The actual Plain Text data is a simple string of ASCII characters.

The Plain Text Extension block is 15 bytes in length and has the following structure:

```
typedef struct _GifPlainTextExtension
{
  BYTE Introducer;          /* Extension Introducer (always 21h) */
  BYTE Label;               /* Extension Label (always 01h) */
  BYTE BlockSize;           /* Size of Extension Block (always 0Ch) */
  WORD TextGridLeft;        /* X position of text grid in pixels */
  WORD TextGridTop;         /* Y position of text grid in pixels */
  WORD TextGridWidth;       /* Width of the text grid in pixels */
  WORD TextGridHeight;      /* Height of the text grid in pixels */
  BYTE CellWidth;           /* Width of a grid cell in pixels */
  BYTE CellHeight;          /* Height of a grid cell in pixels */
  BYTE TextFgColorIndex;    /* Text foreground color index value */
  BYTE TextBgColorIndex;    /* Text background color index value */
  BYTE *PlainTextData;      /* The Plain Text data */
  BYTE Terminator;          /* Block Terminator (always 0) */

} GIFPLAINTEXT;
```


Ex.1007 / Page 473 of 1158
APPLE INC.

Introducer contains the value 21h and is used to identify the start of a Extension data block.

Label contains the value 01h and is used to identify this block of data as a Plain Text Extension.

BlockSize contains the value 0Ch, which is the number of bytes contained in the fields following the BlockSize field.

TextGridLeft and TextGridTop contain the X and Y coordinates (position) of the text grid with respect to the upper-left corner of the display screen (coordinate 0,0).

TextGridWidth and TextGridHeight contain the size of the text grid in pixels.

CellWidth and CellHeight contain the size in pixels of each character cell in the grid.

TextFgColorIndex contains an index into the Global Color Table to retrieve the color of the text.

TextBgColorIndex contains a Global Color Table index value to be used as the color for the background of the text.

PlainTextData contains the actual textual information that is to be rendered as a graphic. This field contains one or more sub-blocks of data. Each sub-block begins with a byte that indicates the number of data bytes that follow. From 1 to 255 data bytes may follow this byte. There may be any number of sub-blocks in this field.

Terminator contains the value zero and marks the end of the Plain Text Extension block.

Application Extension block

Application Extension blocks contain application-specific information in a way similar to the way tags are used in the TIFF and TGA image file formats. Information not normally found in a GIF-format file may be stored in an Application Extension block and then read by any application that understands how to interpret the data. Any number of Application Extension blocks may appear in a GIF file.

Application Extension data is application-readable only. All data stored in this extension is designed to be acted upon by the software application that is reading and processing the GIF data stream. To store human-readable data the Comment Extension block is used instead (see the "Comment Extension block" section).

Ex.1007 / Page 474 of 1158
APPLE INC.

Examples of data stored in an Application Extension block include instructions on changing video modes, applying special processing to displayed image data, and storing additional color tables. Information used to control the computer platform executing the application can also be stored. This can include information on how to manipulate files, how to access peripheral devices such as modems and printers, and how to send audible signals to the audio speaker.

The Application Extension block is 14 bytes in length and has the following structure:

```
typedef struct _GifApplicationExtension
{
  BYTE Introducer;          /* Extension Introducer (always 21h) */
  BYTE Label;               /* Extension Label (always FFh) */
  BYTE BlockSize;           /* Size of Extension Block (always 0Bh) */
  CHAR Identifier[8];       /* Application Identifier */
  BYTE AuthentCode[3];      /* Application Authentication Code */
  BYTE *ApplicationData;    /* Point to Application Data sub-blocks */
  BYTE Terminator;          /* Block Terminator (always 0) */

} GIFAPPLICATION;
```

Introducer contains the value 21h and is used to identify the start of a Extension data block.

Label contains the value FFh and is used to identify this block of data as an Application Extension.

BlockSize contains the value 0Bh, which is the number of bytes in the Identifier and AuthentCode fields.

Identifier may contain up to eight printable 7-bit ASCII characters. These characters are used to identify the application that wrote the Application Extension block. If this identifier value is recognized, the remaining portion of the block is read and its data acted upon. If the identifier value is not recognized, the remaining portion of the block is read and its data is discarded.

AuthentCode contains a value that is used to uniquely identify a software application that created the Application Extension block. This field may contain a serial number, a version number, or a unique binary or ASCII code used to identify an individual software application or computer platform. This field may be used to allow only specific copies or revisions of a particular software application to access the data in certain Application Extension blocks.

Ex.1007 / Page 475 of 1158
APPLE INC.

ApplicationData contains the information that is used by the software application. This field is structured in a series of sub-blocks identical to the data found in a Plain Text Extension block.

Terminator contains the value zero and marks the end of the Application Extension block.

To understand how a GIF reader could interpret Application Extension block information, consider the following example.

An application reading a GIF file comes across an Application Extension. The Identifier field contains the characters "CHKDATE". This identifier is recognized by the application reading the GIF file. The AuthentCode field contains the value "UNX", which is an indication that only versions of this software application running under the UNIX operating system should use the data in this block. All versions of the program not running under UNIX should ignore this block.

The application reading this GIF file knows that a CHKDATE block holds a 2-byte date stamp in the ApplicationCode field. If the current system date is not the same as this date stamp value, the next Graphics Rendering block should not be displayed. The count byte is read from the data sub-block and then the two-byte stamp value. The Terminator field value follows this stamp value.

A second Application Extension is read containing the identifier "CLRSCRN". The application recognizes this identifier and knows that it is an instruction for the display screen to be cleared immediately. The AuthentCode field is not used in this block and its value is read and ignored. This block also does not contain any data sub-blocks, and therefore the block terminator value occurs immediately.

A third Application Extension is read containing the identifier "SOUNDBYT". This identifier informs the application that this block contains audio data that should be sent to the sound card driver installed in the system. The AuthentCode field contains the code "CDI", which indicates the format of the audio data stored in this block. If the AuthentCode field is recognized, then the data sub-blocks are read, and the data is sent to the computer platform's audio system until a zero count byte is read.

Finally, a fourth Application Extension is read containing the identifier "SPECIAL". This particular identifier is not recognized by the application reading the GIF file, so the AuthentCode field and the ApplicationData field are read and ignored.

Ex.1007 / Page 476 of 1158
APPLE INC.

The above examples are only a few of the hundreds of ways Application Extension blocks may be used to provide control over a computer system and the way GIF images are displayed. The GIF89a specification does not list any specific examples of the use of Application Extension blocks, nor does it include a standard list of identifiers; presumably, this is left up to the ingenuity of the developer.

Comment Extension block

The Comment Extension block is used to insert a human-readable string of text into a GIF file or data stream. Each comment may contain up to 255 7-bit ASCII characters, including all the ASCII control codes. Any number of Comment Extension blocks may occur in a GIF file, and they may appear anywhere after the Global Color Table. It is suggested, however, that comments should appear before or after all image data in the GIF file.

All data stored in the Comment Extension is designed to be read only by the human user examining a GIF file or data stream. All comment data should be ignored by the application reading the GIF data stream. To store computer-readable data and instructions, use the Application Extension block. (See the section called "Application Extension block" earlier in this article.)

Comments are typically used to identify the source of the GIF image, its author, the creating software, the creation time and date, the copyright notice for the image data, and so on. Several image display programs that accommodate version 89a images also have the capability of displaying comment data stored within the GIF files.

Comment Extension blocks must always remain independent of all other data in a GIF file. Comment Extension data is not modified by the information in any other Extension blocks, and comments should not contain data that is intended to be read and interpreted as instructions by software applications.

The Comment Extension block may vary from 5 to 259 bytes in length and has the following structure:

```
typedef struct _GifCommentExtension
{
  BYTE Introducer;    /* Extension Introducer (always 21h) */
  BYTE Label;         /* Comment Label (always FEh) */
  BYTE *CommentData; /* Pointer to Comment Data sub-blocks */
  BYTE Terminator;    /* Block Terminator (always 0) */

} GIFCOMMENT;
```


Ex.1007 / Page 477 of 1158
APPLE INC.

Introducer contains the value 21h and is used to identify the start of a Extension data block.

Label contains the value FEh and is used to identify this block of data as a Comment Extension.

CommentData contains one or more sub-blocks of ASCII string data. The character strings stored in the CommentData field sub-blocks are not required to be NULL-terminated.

Terminator contains the value 0 and marks the end of the Comment Extension block. The value of the Terminator field may be used as a NULL-terminator if "size + 1" bytes of comment data is read from the block.

For Further Information

For further information about GIF, see the specifications included on the CD-ROM that accompanies this book:

CompuServe Incorporated, *GIF Graphics Interchange Format: A standard defining a mechanism for the storage and transmission of bitmap-based graphics information.* Columbus, OH, 1987.

CompuServe Incorporated, *Graphics Interchange Format: Version 89a.* Columbus, OH, 1990.

You can also obtain a copy of the GIF89a specification from many BBSs and online services, or directly from CompuServe at:

CompuServe Incorporated
Attn: Graphics Technology Department
5000 Arlington Center Boulevard
Columbus, OH 43220
Voice: 614-457-8600
Voice: 800-848-8199
WWW: *http://www.compuserve.com/*

Several packages included on the CD-ROM display and convert GIF images.

Ex.1007 / Page 478 of 1158
APPLE INC.

NAME:	GRASP
ALSO KNOWN AS:	Graphical System for Presentation, .GL, .CLP, .FNT, .PIC, .SET, TXT
TYPE:	Animation
COLORS:	256
COMPRESSION:	RLE
MAXIMUM IMAGE SIZE:	Variable
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Microtex Industries
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	GRASP
SPECIFICATION ON CD:	Yes (summary description)
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	Microsoft RIFF, FLI

USAGE: GRASP is a simple animation format capable of displaying low- and medium-resolution images, text, and simple sounds.

COMMENTS: At one time the most widely used animation format around, GRASP's fortunes have been declining, perhaps because of increased interest in digital video formats such as AVI and QuickTime.

Overview

GRASP (GRAphical System for Presentation) is an MS-DOS application used to create and play back simple animated sequences. Such animations are incorporated into other applications, such as graphical presentations, educational tutorials, and games.

GRASP is a simple toolkit of utilities used to create and play back animations. The basic tool found in GRASP is the editor (*GRASP.EXE* under MS-DOS). The editor program organizes graphics and command information together into a GRASP animation. An animation may contain both text and images, which can be presented in a variety of ways. Simple sounds may be generated from the

Ex.1007 / Page 479 of 1158
APPLE INC.

PC's speaker, and the animation may be controlled via user input. GRASP supports all standard EGA and VGA display modes.

Most GRASP animations that you will encounter are stored in a GRASP library file (with a .GL extension). A .GL file is actually a library of separate files that contain information required to display a GRASP animation. These files are normally stored as separate disk files when used by GRASP. But when an animation must be transported to another environment or incorporated into another application, the GRASP Library Manager (*GLIB.COM* under MS-DOS) combines all the necessary files into a single .GL library file.

A GRASP library may contain four different types of information files; the formats of these files are discussed later in this article. When these files are stored separately on disk, their types can be identified by their file extensions:

.TXT	Command file
.PIC	Picture file
.CLP	Clip file (picture file without a color map)
.FNT	Font character information (also *.SET)
.SET	Font character information

A GRASP animation is played back using the GRASP run-time display program (*GRASPRT.EXE* under MS-DOS). All applications playing back .GL animation files will call this program to display the animation. The GRASPRT program is capable of reading the animation from a single .GL file or from each information file stored separately on disk.

A .GL file is created as follows:

1. Using a paint or imaging program, create or capture the individual frames you want animated. The images must be saved to disk using the PCPAINT/Pictor file format.
2. Use the GRASP editor program to create a sequence of commands that will be used to display the animation.
3. Combine all the resulting data files into a single .GL file using the GRASP Library Manager. The resulting .GL file may be played back using the GRASP run-time display engine.

Ex.1007 / Page 480 of 1158
APPLE INC.

File Organization

All GRASP library files begin with a header. The header is a directory of the separate files stored in the .GL file. The header varies in size depending on the number of files stored. The format of the header is shown below:

```
typedef struct _GLHeader
{
  WORD DirectorySize;          /* Size of header in bytes */
  struct _FileEntry
  {
    DWORD FileOffset;          /* Offset of file in the .GL file */
    CHAR  FileName[13];        /* Name of the file */
  } FileDirectory[DirectorySize];

} GLHEADER;
```

DirectorySize is the size of the .GL header in bytes.

The FileDirectory contains one FileEntry structure for each file stored in the .GL.

FileOffset contains the offset location of the file within the .GL file. If a FileOffset value is 00h, then the end of the file directory has been reached.

FileName contains the original disk filename of the file. Each file in a .GL library file is stored as a 4-byte value indicating the size of the file, followed by the file data itself. A structure representing this arrangement appears as follows:

```
typedef struct _FileData
{
  DWORD FileLength;            /* Size of the file data in bytes */
  BYTE  FileData[FileLength];  /* File data */

} FILEDATA;
```

File Details

This section describes the types of files contained in a GRASP library.

Command File

A command (.TXT) file is simply a script of GRASP commands read by the .GL playback program and used to display the animation. A command file is an ordinary ASCII text file and usually begins with a few comment lines identifying the name of the file and author, title of the animation, date of creation, and so on. There is one command file per .GL file.

Ex.1007 / Page 481 of 1158
APPLE INC.

A number of commands allow GRASP animations to have many special operations and effects. Some of the commands are:

- Load image files and fonts
- Execute MS-DOS programs
- Receive input from the user
- Draw geometric shapes
- Change video modes
- Change color maps
- Create sound
- Display text
- Perform fades

Each command is a keyword followed by zero or more arguments delimited by commas. All commands are case-insensitive except for literal strings which must appear in double quotes. All comments begin with a semicolon and are ignored during playback.

Consider the following example. A .GL file which contains a simple, five-frame animation that displays in an infinite loop might contain the following command file:

```
video L                 ; Set video mode to 320x200x256 VGA
pload PALETTE,1         ; Load palette.pic into picture buffer 1
palette 1               ; Set the palette
pfree 1                 ; Free picture buffer 1
cload CLP0,1            ; Load clp0.pic into buffer 1
cload CLP1,2            ; Load clp1.pic into buffer 2
cload CLP2,3            ; Load clp2.pic into buffer 3
cload CLP3,4            ; Load clp3.pic into buffer 4
cload CLP4,5            ; Load clp4.pic into buffer 5
forever:               ; Label
putup 80,50,1,10        ; Display buffer 1
putup 80,50,2,10        ; Display buffer 2
putup 80,50,3,10        ; Display buffer 3
putup 80,50,4,10        ; Display buffer 4
putup 80,50,5,10        ; Display buffer 5
putup 80,50,5,10        ; Display buffer 5
putup 80,50,4,10        ; Display buffer 4
putup 80,50,3,10        ; Display buffer 3
putup 80,50,2,10        ; Display buffer 2
goto forever            ; GOTO 'forever' label (and repeat
                        ; display of buffers)
```


Ex.1007 / Page 482 of 1158
APPLE INC.

Image Files

Each frame in a GRASP animation is stored using the Pictor PC Paint file format. Files with the .PIC extension contain a color map, whereas files with the .CLP extension do not. It is common for a GRASP animation to include a single .PIC file that contains only a color map and to store all of its animation frames as mapless .CLP files.

For more information about the Pictor PC Paint file format, refer to the article about it later in this book.

Font File

Text is displayed in a GRASP animation by first loading a font and then displaying it using the *text* command. Fonts are stored with the extension .FNT or .SET. A font file may contain data for up to 256 font characters. The header for the font file is shown below:

```
typedef struct _FontHeader
{
  WORD FileLength;              /* Size of font file in bytes */
  BYTE NumberOfCharacters;      /* Number of characters in file */
  BYTE FirstCharacterValue;     /* ASCII value of first character */
  BYTE CharacterWidth;          /* Width of character in pixels */
  BYTE CharacterHeight;         /* Height of character in pixels */
  BYTE CharacterSize;           /* Size of character in bytes */

} FONTHEADER;
```

FileLength is the size of the font file in bytes, including header.

NumberOfCharacters is the number of font character defined in this file. This number may be 0 through 255, with 0 representing 256 characters stored.

FirstCharacterValue is the ASCII value of the first font character appearing in the data (font characters are arranged in ASCII value order). Most fonts start with the space character (ASCII value 20h).

CharacterWidth and CharacterHeight represent the size of the font in pixels.

CharacterSize is the number of bytes required to store each font character. This value is usually calculated from the pixel size of the font (CharacterWidth×CharacterHeight / 8).

The font data immediately follows the header. Each character is stored by row, and each row is padded out to the nearest byte boundary. An 8×16 font contains sixteen rows by eight columns of 1-bit pixels. A 0 bit indicates black and a 1 bit indicates color.

Ex.1007 / Page 483 of 1158
APPLE INC.

An 8×16 font requires 16 bytes of data per character to store. A full 256-character font is then 4096 bytes in size. The maximum size of a font file is 64K.

For Further Information

For further information about GRASP, see the article included on the CD-ROM that accompanies this book.

GRASP was originally created by Microtext Incorporated of Irvine, CA. It was bought in 1988 by Paul Mace software, where it is maintained today. Information on GRASP and GRASP Multimedia may be obtained directly from Paul Mace Software:

Paul Mace Software, Inc.
Attn: Steven Belsky
400 Williamson Way
Ashland, Oregon 97520
Voice: 503-488-2322
FAX: 503-488-1549
BBS: 503-482-7435
WWW: *http://www.pmace.com/pms.htm*

Ex.1007 / Page 484 of 1158
APPLE INC.

NAME:	GRIB
ALSO KNOWN AS:	Gridded Binary
TYPE:	Various
COLORS:	NA
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	Binary bit-oriented
ORIGINATOR:	World Meteorological Organization
PLATFORM:	All
SUPPORTING APPLICATIONS:	Unknown
SPECIFICATION ON CD:	Yes (summary description)
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	BUFR

USAGE: Transfer and transmission of weather and other data.

COMMENTS: The GRIB format is outside the scope of this book, but we include a brief description because it is likely to be more useful in the future as interest in geographical information systems increases.

Overview

GRIB was created by the World Meteorological Organization (WMO) and is officially designated as FM 92-VIII Ext. GRIB (GRIdded Binary). It is designed to support fast computer-to-computer transmission of large volumes of data. Speed and efficiency are the key words here.

The format documentation is subtitled: "The WMO Format for the Storage of Weather Product Information and the Exchange of Weather Product Messages in Gridded Binary Form." Data in GRIB files, as the name suggests, is expected to be in gridded form, that is, arrayed in a rectilinear fashion. That this suggests our idea of a bitmap is no coincidence, although the WMO and its affiliates normally use the format for the transmission of observational data such as air pressure and temperature.

Ex.1007 / Page 485 of 1158
APPLE INC.

GRIB data streams and files adhere to the specification called "WMO Standard Formats for Weather Data Exchange Among Automated Weather Information Systems."

File Organization

GRIB files consist of a number of records, each of which may contain the following information:

- Indicator section
- Product definition section (PDS)
- Optional grid description section (GDS)
- Otional bitmap section (BMS)
- Binary data section (BDS)
- ASCII characters 7777

File Details

Detailing the internals of GRIB is beyond the scope of this article. GRIB is extremely complex and is, at this point, used in a narrow area of technology. This, coupled with the fact that the document is written in a dialect of government-ese, makes it tough sledding even for initiates. Nevertheless, if you need to understand the GRIB format, the document included on the CD-ROM should get you started. Good luck. Please note that the people who are responsible the GRIB documentation were as nice as could be.

For Further Information

For detailed information about GRIB, see the paper included on the CD-ROM that accompanies this book:

> Stackpole, John D., "The WMO Format For the Storage of Weather Product Information and the Exchange of Messages in Gridded Binary Form."

You can also get information from the University of Wisconsin NMS homepage:

> *http://java.meteor.wisc.edu/*

Ex.1007 / Page 486 of 1158
APPLE INC.

Additional information on WMO data specifications can also be found in the following document:

Standard Formats for Weather Data Exchange Among Automated Weather Information Systems, Document Number FCM-S2-1990.

This document is available from:

U.S. Department of Commerce/National Oceanic and Atmospheric Administration (NOAA)
Attn: Ms. Lena Loman
Office of the Federal Coordinator for Meteorological Services and Supporting Research (OFCM)
6010 Executive Blvd, Suite 900
Rockville, MD 20852
Voice: 301-443-8704

For more information about the GRIB format, contact:

U.S. Department of Commerce/National Oceanic and Atmospheric Administration (NOAA)
National Meteorological Center
Attn: Dr. John D. Stackpole
Chief, Production Management Branch, Automation Division
WINMC42, Room 307, WWB
5200 Auth Road
Camp Springs, MD 20746
Voice: 301-763-8115
FAX: 301-763-8381
Email: *jstack@sun1.wwb.noaa.gov*

Ex.1007 / Page 487 of 1158
APPLE INC.

Harvard Graphics

NAME:	Harvard Graphics
ALSO KNOWN AS:	None
TYPE:	Metafile
COLORS:	NA
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Software Publishing
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Harvard Graphics, other presentation graphics
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Proprietary to Software Publishing; used by Harvard Graphics business graphics application.

COMMENTS: Software Publishing considers the format proprietary, but will consider a license arrangement.

Overview

Software Publishing, the originator of the Harvard Graphics format, considers this format to be proprietary. Although we wish this were not the case, we can hardly use our standard argument—that documenting and publicizing file formats make sales by seeding the aftermarket. Harvard Graphics has been the top, or one of the top, sellers in the crowded and cutthroat MS-DOS business graphics market, and has remained so despite the lack of cooperation of Software Publishing with external developers.

While we would be happy to provide information about the format if it were available, we have failed to find any during our research for this book, so it appears that Software Publishing has so far been successful in their efforts to restrict information flow from their organization.

Ex.1007 / Page 488 of 1158
APPLE INC.

To be fair, Software Publishing appeared to consider our request to include the Harvard Graphics format in this book. Although the organization wishes to continue to exert some measure of control over information about the format, they are willing to enter into license arrangements with third-party developers that include nondisclosure agreements. We got the impression that reasonable requests would not be refused.

For Further Information

To obtain further information about obtaining the Harvard Graphics file format under distribution license, contact Software Publishing at:

Software Publishing Inc.
P.O. Box 54983
3165 Kifer Road
Santa Clara, CA 95056-0983
Voice: 408-986-9800

Ex.1007 / Page 489 of 1158
APPLE INC.

Hierarchical Data Format

NAME:	Hierarchical Data Format
ALSO KNOWN AS:	HDF
TYPE:	Metafile
COLORS:	NA
COMPRESSION:	NA
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	NA
ORIGINATOR:	National Center for Supercomputer Applications (NCSA)
PLATFORM:	All
SUPPORTING APPLICATIONS:	Various
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	FITS

USAGE: Transport and exchange of scientific data, including images, between different applications and platforms.

COMMENTS: A tremendously versatile format that supports the inclusion of various types of "metadata" while continuing to provide support for more mundane data objects, such as images. Used by applications associated with scientific visualization, and well-supported by a portable library of functions from the NCSA Software Tools Group.

Overview

The Hierarchical Data Format (HDF) was created by a group at the National Center for Supercomputer Applications (NCSA) to support the needs of the scientific community with respect to scientific data management. The format was designed to provide support for the following:

Ex.1007 / Page 490 of 1158
APPLE INC.

- Scientific data and metadata
- Multiple diverse platforms
- Support for a range of software tools
- Rapid and efficient data transfer
- Extensibility

The format is, in the words of Mike Folk at NCSA, a "self-describing extensible file format based on the use of tagged objects that have standard meanings." The specification is extremely complete and is included on the CD, so we will confine ourselves to some introductory remarks. We include the format in this book because we feel that data visualization will be increasingly important in the future.

HDF supports lower-level data types such as multidimensional gridded data, 2D and 3D bitmap images, polygonal mesh data, multivariate datasets, sparse matrices, finite element (FE) data, spreadsheets, splines, non-Cartesian coordinate data, and text.

The file structure is entirely tag-based and is founded on the assumption that the needs of scientists are unknowable in advance. While this might result in anarchy in another situation, manipulation of data associated with public tags is supported by a portable, publicly available library maintained by NCSA. Note that the HDF Project at NCSA is closely associated with the NCSA Software Tools group, which works to support the scientific community, especially in the area of scientific visualization.

Tags are grouped under the unifying concept of a Vset, which is a hierarchical grouping structure flexible enough to support multiple views, useful for data analysis and retrieval.

For Further Information

For further information about the HDF format, see the specification included on the CD-ROM that accompanies this book. You can also contact NCSA, the organization responsible for maintaining the spec, at:

National Center for Supercomputer Applications
Attn: Michael Folk
University of Illinois
605 East Springfield Avenue

Ex.1007 / Page 491 of 1158
APPLE INC.

Champaign, IL 61820
Voice: 217-244-0072
FAX: 217-244-1987
Email: *mfolk@ncsa.uiuc.edu*

The latest HDF specification is also available via FTP. You must sign a license and then download from

ftp://ftp.ncsa.uiuc.edu/Documentation/HDF/HDF3.2/
ftp://ftp.ncsa.uiuc.edu/Documentation/HDF/HDF.Vset2.1/

Other online information resources include:

http://hdf.ncsa.uiuc.edu:8001/
http://www.ncsa.uiuc.edu/SDG/Software/HDF/HDFIntro.html

Ex.1007 / Page 492 of 1158
APPLE INC.

NAME:	IFF*
ALSO KNOWN AS:	Interchange File Format, ILM, ILBM, LBM, Amiga Paint
TYPE:	Bitmap
COLORS:	1- to 24-bit
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	64K×64K pixels
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Electronic Arts, Inc., Commodore-Amiga, Inc.
PLATFORM:	Amiga, MS-DOS, others
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	Microsoft RIFF, GIF, PNG

USAGE: On platforms other than the Amiga, IFF is used mostly for storing image and sound data. On the Amiga almost any type of data may be found in an IFF file. The file extension usually indicates the type of data stored in an IFF file.

COMMENT: IFF is an older data file format found on most every system. Its versatility has not been greatly utilized outside of the Amiga platform.

Overview

IFF (Interchange File Format) is a general purpose data storage format that can associate and store multiple types of data. IFF is portable and has many well-defined extensions that support still-picture, sound, music, video, and textual data. Because of this extensibility, IFF has fathered a family of special purpose file formats all based on IFF's simple data structure.

* Our thanks to Jolyon Ralph and Ernie Wright for their contributions to this article.

Ex.1007 / Page 493 of 1158
APPLE INC.

IFF is most often associated with the Commodore-Amiga computer and originated on that system. IFF is fully supported by the Amiga operating system and is used for storing virtually every type of data found in the Amiga's filesystem. Initialization files, documents, temporary data, and data exported from the clipboard may all be stored using the IFF format.

The most common IFF family member is ILBM, or InterLeaved BitMap. ILBM files are the standard image file format for the Commodore-Amiga computer and are the type of IFF files with which most graphics people are familiar.

IFF files are common in the MS-DOS and UNIX environment as well and usually have the file extension .IFF or .LBM. Electronic Arts' DeluxePaint program is generally credited with making IFF known to the MS-DOS community. For a time IFF was a widely accepted 24-bit format under MS-DOS, but it was eventually replaced first by TIFF and TGA, and then by JFIF.

IFF faces compatibility problems when the occasional program fails to write IFF file data using the big-endian byte order. This prevents most programs from reading these IFF files. Other problems created by bad IFF file writers include writing planar image data improperly and failing to use only linefeeds to terminate lines of text. Unfortuantely, some people (those who are all too willing to shoot the messenger), have blamed IFF, rather than bad software, for these problems.

Today IFF is a widely used format that is supported by most graphics programs found on MS-DOS, MS Windows, Macintosh, UNIX, and Amiga systems. The format basically remained unchanged since its specification was released in 1985, but many extensions to the format have been created and documented by a great many software developers, making IFF one of the most utilized data file formats of today.

File Organization

IFF files are constructed entirely of chunks. A chunk is a data structure containing a 4-byte ID, a 4-byte size value, and possibly a block of data. Each chunk is the same, simple structure and differs only in the data it contains. You can think of a chunk as an envelope or wrapper that identifies a collection of data.

The data stored in a chunk can be anything: graphics, sound, animations, text from a word processor, or a collection of 3D objects. Any kind of data can be stored in a chunk, including a chunk itself.

Ex.1007 / Page 494 of 1158
APPLE INC.

Nesting one or more chunks within a chunk is common in IFF files. In fact, an IFF file is conceptually nothing more than a single chunk containing one or more other chunks as data. There is also no specified limit as to the number of nesting levels within a chunk.

IFF nesting is a powerful organizing principle. It offers the same sort of organizational advantages that nested directories and subdirectories do for filesystems. The down side of nesting is that it introduces a certain amount of complexity that can make IFF appear difficult to interpret.

Figure IFF-1 illustrates the "chunks within a chunk" concept.

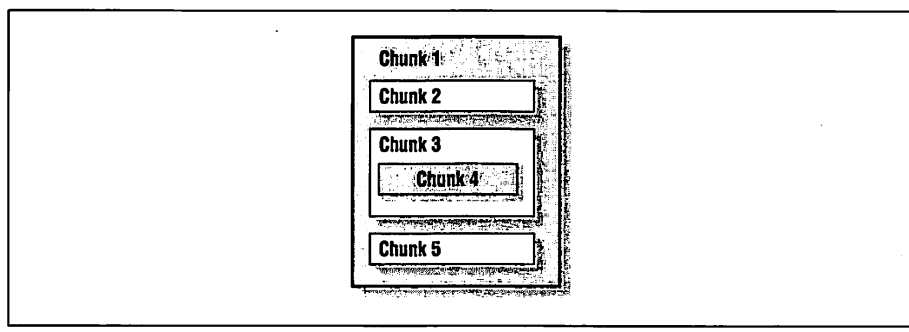


FIGURE IFF-1: *A chunk file structure*

Most IFF files contain a single chunk called a FORM chunk. This chunk stores the formatting and identification information for all chunks and data in the IFF file. All other chunks in the file are stored within the FORM chunk. The basic structure of the FORM chunk is illustrated in Figure IFF-2.

In this example, the FORM chunk is of type ILBM (InterLeaved BitMap) and contains three chunks, each of which contain data blocks that together define an image. The FORM ILBM is the most common file type for storing still-picture graphical data in an IFF file. "ILBM" is the type identifier for the FORM. It tells readers what kind of FORM chunk this is and what chunks might be expected within it. The three chunks in this example are the BMHD (BitMap HeaDer), CMAP (Color MAP), and BODY (the actual pixels).

Another common FORM type is 8SVX (8-bit Sampled VoX, or Voice), a format for digitized sound samples. A simple FORM 8SVX is shown in Figure IFF-3.

Ex.1007 / Page 495 of 1158
APPLE INC.

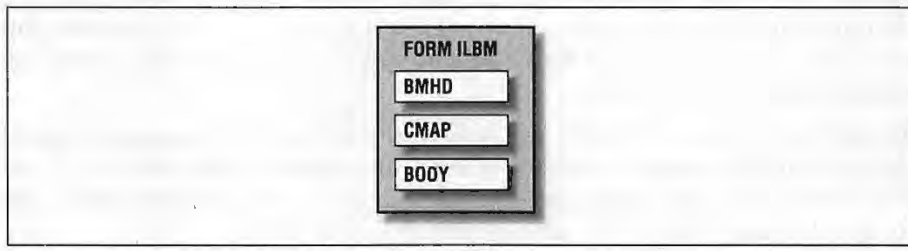


FIGURE IFF-2: *FORM ILBM file structure*

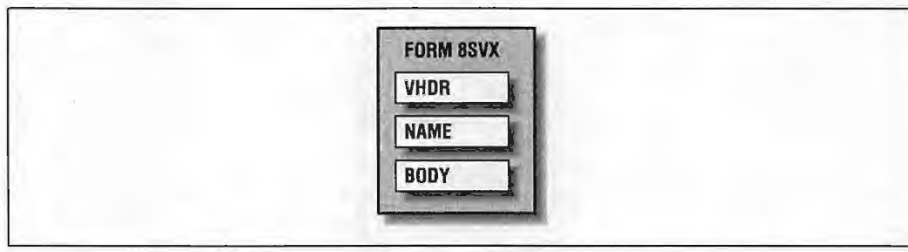


FIGURE IFF-3: *FORM 8SVX file structure*

As you can see, Figures IFF-2 and IFF-3 aren't very different. The high-level structure of all IFF files is similar because it is created from the same simple chunking rules, regardless of what kind of data is stored in the files.

IFF files that contain a single FORM chunk are by far the most common. In fact, if you confine yourself to the most widely used FORM types such as ILBM, you may never encounter any other IFF structure. But group structures do exist that allow IFF writers to collect multiple FORMs into a single file.

A CAT chunk is used to append or "concatenate" two or more FORM chunks together in a single IFF file. Figure IFF-4 shows a CAT ILBM file that contains two FORM ILBM chunks.

In this figure, the CAT chunk contains a single ILBM chunk that contains an ILBM type ID that identifies the type of data stored in each FORM chunk. These two FORM chunks have the same format they would have if they were stored in separate IFF files.

Ex.1007 / Page 496 of 1158
APPLE INC.



FIGURE IFF-4: *CAT ILBM file structure*

All of the FORM chunks in a CAT chunk need not store the same type of data. Figure IFF-5 illustrates a CAT chunk that stores a FORM ILBM chunk and a FORM 8SVX chunk.

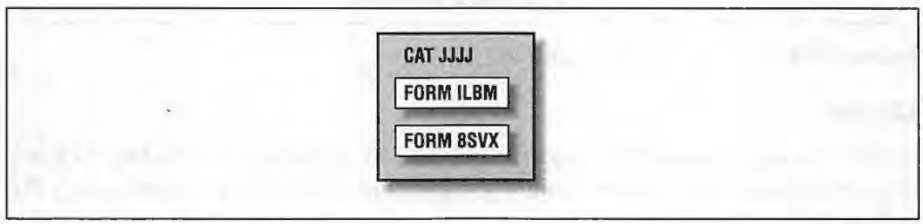


FIGURE IFF-5: *CAT JJJJ file structure*

When the FORM chunks in a CAT chunk do not all store the same type of data, a contents type identifier of "JJJJ" is used to indicate that the CAT contains FORMs of more than one type, or that the IFF writer did not care what type(s) of FORM might be in the file. CAT type IDs are often referred to as "hints," because each FORM unambiguously identifies what it contains.

LIST chunks also allow the storage of multiple data objects within a single IFF file but add the ability to group data objects together and have them share common data by the use of the PROP (property) chunk. Figure IFF-6 illustrates a LIST ILBM file that contains two images that share a common bitmap header and color map by using a PROP chunk.

File Details

This section describes the details of IFF chunks.

Ex.1007 / Page 497 of 1158
APPLE INC.

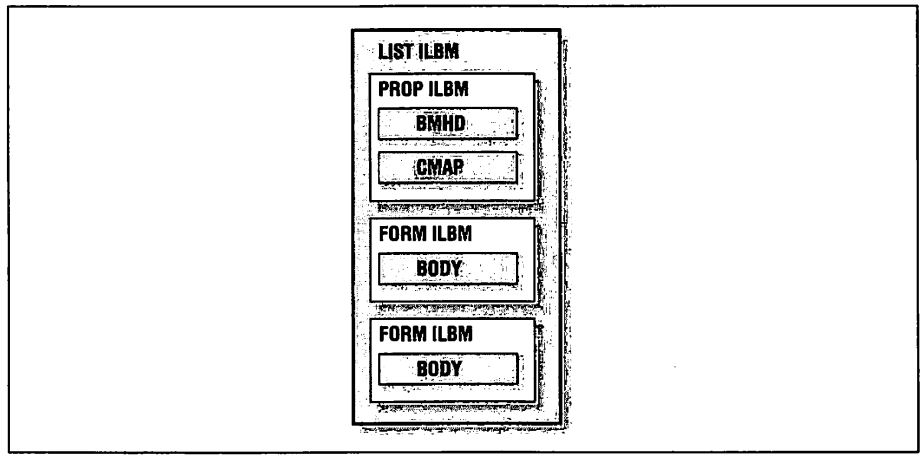


FIGURE IFF-6: *LIST ILBM file structure*

Chunks

All IFF files are composed of very simple data structures called *chunks.* A chunk may be anywhere from four to eight gigabytes in size and is represented by the following data structure:

```
typedef struct _Chunk
{
    char  ChunkId[4];
    DWORD Size;
    BYTE  Data[];
} CHUNK;
```

ChunkId is the ASCII identifier of the chunk. Identifiers are always alphanumeric characters, and they are right-padded with spaces if they contain fewer than four characters. FORM Type IDs may only use uppercase characters, and Chunk IDs may be mixed case.

Size is the number of data bytes stored in the Data field. This value does not include the presence of a padding byte that may follow the data. If the chunk contains no data, then this value is 0.

Data is the actual chunk data. The number of bytes of data stored in this field is indicated by the value in the Size field.

Ex.1007 / Page 498 of 1158
APPLE INC.

We can use the _Chunk structure to show the nested chunks within a FORM ILBM file:

```
typedef struct _ChunkFORM_ILBM
{
   char  ChunkId[4];          /* "FORM" */
   DWORD Size;                /* FORM size (size of file minus 8) */

   /* Start of FORM chunk's data */

   char  TypeID[4];           /* "ILBM" */

   struct _ChunkBMHD
   {
      char  ChunkId[4];       /* "BMHD" */
      DWORD Size;             /* Size of Data */
      BMHD  Data;             /* Bitmap header data */
   }
   struct _ChunkCMAP
   {
      char  ChunkId[4];       /* "CMAP" */
      DWORD Size;             /* Size of Data */
      CMAP  Data;             /* Color map data */
   }
   struct _ChunkBODY
   {
      char  ChunkId[4];       /* "BODY" */
      DWORD Size;             /* Size of Data */
      BODY  Data[];           /* Image data */
   }
}
```

This is the C code version of Figure IFF-2. The file contains a single FORM chunk, which like all chunks begins with a 4-character chunk ID and a 4-byte size. The data for the FORM chunk begins with a 4-character "FORM Type Identifier," or type ID, that identifies the kind of object stored in the FORM. For ILBMs the type ID is just "ILBM". The type ID is followed by a collection of chunks that describe the ILBM, including the bitmap header, color map, and pixel bits.

(Note that we're illustrating the structure of a FORM ILBM using C code here, but you shouldn't actually use a structure like this in your IFF code. The contents of a FORM chunk vary too much to be captured by a single C structure. Nor should you assume that these are the only chunks in a FORM ILBM, or that they will occur in this particular order.)

Chunks must always begin on an even byte boundary. If a chunk contains an odd number of data bytes, then the chunk that follows would improperly begin

Ex.1007 / Page 499 of 1158
APPLE INC.

on a odd-byte boundary. To preserve alignment, a padding byte will be added between the odd-length data field of a chunk and the Chunk ID of the next chunk. This padding byte always has a value of zero and is not considered to be part of the chunk data. If the Size field of a chunk contains an odd value, then you should assume that a padding byte is present.

Parsing an IFF file is a process of reading chunk identifiers, using the data in known chunks and skipping over unknown chunks. The chunk identifiers are therefore crucial in determining whether the IFF file reader recognizes the format of the data stored in a chunk.

Chunk Content Identifiers

The first four bytes of every chunk identifies the format of the chunk's data. These bytes are called the Chunk Content Identifier, or Chunk ID for short. Chunk IDs are made of ASCII characters in the range 0x20 to 0x7E (" " to "~"). The restriction to uppercase and no punctuation applies only to FORM type IDs. Spaces are used only to pad out IDs that are fewer than four printable characters in length.

Each ID represents a specific format of data. If an IFF reader does not recognize the ID of a chunk, then the reader cannot know the format of the chunk's data, and therefore should skip over the chunk.

Five primary chunk IDs are reserved by the IFF specification. They are "FORM", "LIST", "PROP", "CAT ", and "   ". Note that the CAT chunk contains a single padding space, and the ID of the Filler Chunk is all spaces.

Chunk IDs can also be used to indicate the revision level of a chunk. For example, the revision IDs for the FORM chunk are "FOR1", FOR2", "FOR3", and so on. For the CAT chunk, the revisions are "CAT1", "CAT2", "CAT3", and so on. The FORM, LIST, and CAT chunks each have nine revisions reserved by the IFF-85 specification, bringing the total number of reserved chunk IDs to 32.

If an IFF reader does not recognize the FORM Type ID of a FORM, CAT or LIST chunk, it may continue reading to find any nested FORM chunks in the file that it does recognize. If the first chunk in the file does not have a Group Type ID of FORM, CAT, or LIST, then the reader should assume that it is not an IFF-format file.

The number and kind of data chunks that appear in a FORM, and the order in which they appear, is determined by the FORM type ID. Some data chunks

Ex.1007 / Page 500 of 1158
APPLE INC.

must always be present, such as those required of a BMHD chunk in a FORM ILBM. Some chunks must appear in a specific order; for example, a CMAP chunk must always appear before its corresponding BODY chunk. But most data chunks have no restriction on the order in which they occur, or even the number of times that they may occur, in the IFF file.

The Filler Chunk (Chunk ID " ") is a special-purpose data chunk that is used only to provide alignment between chunks in a file. The data stored in a Filler Chunk is meaningless and is never used. For example, a file reader may be designed to read 1K blocks of data from a file stream. In this case, you might require that each chunk in an IFF file be aligned on the closest 1024-byte boundary. Filler chunks would be inserted between all other chunks to provide boundary alignment padding as needed.

To give you an idea of all of the possible data chunks that might be found in a single FORM Chunk, here is a list of a few chunk IDs that are associated with storing textual data:

(C)	Copyright notice and license
ANNO	Annotation or comment
AUTH	Author name
DOC	Document formatting information
FOOT	Footer information of a document
HEAD	Header information of a document
PAGE	Page break indicator
PARA	Paragraph formatting information
PDEF	Deluxe Print page definition
TABS	Tab positions
TEXT	Text for a paragraph
VERS	File version

Here is a listing of some FORM types to give you an idea of what kind of data is stored using the IFF format:

Graphical

ACBM	Amiga Contiguous Bitmap (Microsoft Basic for the Amiga)
DEEP	IFF Deep (24-bit color image)
DR2D	2D object standard format (vector data)
FNTR	Raster font

Ex.1007 / Page 501 of 1158
APPLE INC.

FNTV	Vector font
ILBM	InterLeaved Bitmap (interleaved planar bitmap data)
PICS	Macintosh picture
RGB8	24-bit color image (Impulse)
RGBN	12-bit color image (Impulse)
TDDD	Turbo 3d rendering data (3D objects)
YUVN	YUV image data (V-Lab)
Animation	
ANBM	Animated bitmap
ANIM	Cel animations
SSA	Super smooth animation (ProDAD)
Video	
VDEO	Deluxe Video Construction Set video
Sound	
8SVX	8-bit sampled voice
AIFF	Audio interchange file format
SAMP	Sampled sound
UVOX	Uhuru Sound Software Macintosh voice
Music	
GSCR	General use musical score
SMUS	Simple musical score
TRAK	MIDI music data
USCR	Uhuru Sound Software musical score
Text	
FTXT	Formatted text
WORD	Pro-write word processing

Probably several hundred FORM types and chunk IDs have been created and (we hope) documented by software developers over the past ten years. Most Amiga software archives contain the specifications for many of these chunks. In the section below, we look at the most common IFF data type for storing graphics data, ILBM.

ILBM Chunk

The ILBM format is the most commonly used chunk format for storing graphics data in an IFF file. The only data chunk required to appear in an ILBM is a Bitmap Header chunk (BMHD). It may seem strange not to require the

Ex.1007 / Page 502 of 1158
APPLE INC.

presence of a BODY chunk to store the image data, but an IFF colormap file is a FORM ILBM that contains only a BMHD and colormap (CMAP) chunk but no image data.

ILBMs may optionally contain a colormap (CMAP chunk), hot spot information (GRAB chunk), destination merge data (DEST chunk), or sprite information (SPRT chunk). They may specify a Commodore-Amiga viewport mode (CAMG chunk) or image data (BODY chunk). All of these chunks must appear after the BMHD chunk and before the BODY chunk. BODY must always appear last.

Bitmap Header (BMHD) chunk

The BMHD chunk contains information defining the metrics of the image data. This chunk is always 36 bytes in length and has the following format:

```
typedef struct _BitMapHeaDer
{
    char  ChunkId[4];    /* Chunk Identifier "BMHD" */
    DWORD Size;          /* Size of chunk data in bytes */
    /* Chunk data starts here */
    WORD  Width;         /* Width of image in pixels */
    WORD  Height;        /* Height of image in pixels */
    WORD  Left;          /* X coordinate of image */
    WORD  Top;           /* Y coordinate of image */
    BYTE  Bitplanes;     /* Number of bitplanes */
    BYTE  Masking;       /* Type of masking used */
    BYTE  Compress;      /* Compression method use on image data */
    BYTE  Padding;       /* Alignment padding (always 0) */
    WORD  Transparency;  /* Transparent background color */
    BYTE  XAspectRatio;  /* Horizontal pixel size */
    BYTE  YAspectRatio;  /* Vertical pixel size */
    WORD  PageWidth;     /* Horizontal resolution of display device */
    WORD  PageHeight;    /* Vertical resolution of display device */
} BITMAPHEADERCHUNK;
```

ChunkId contains the chunk content identifier "BMHD".

Size is the number of bytes following the Size field and is always 28.

Width and Height are the width and height, respectively, of the image data in pixels.

Left and Top are the X and Y coordinates position of the upper-left corner of the image. The default values for these fields are 0 and 0.

Bitplanes is the number of bits per pixel used to store the image data.

Ex.1007 / Page 503 of 1158
APPLE INC.

Masking indicates the type of masking used to display the image. Valid values for this field are 0 (standard opaque rectangular image), 1 (mask data is interleaved with image data as an extra bitplane), 2 (pixels that match the value in the Transparency field are transparent), and 3 (image may be lassoed, as in MacPaint).

Compress indicates whether the image data in the BODY chunk is compressed. A value of 0 indicates no compression, and a value of 1 indicates that the data is compressed using the Packer algorithm defined by the IFF-86 ILBM specification.

Padding is used to maintain alignment padding within the BMHD structure and always contains the value zero.

Transparency is a value used with Masking to determine which pixels (if any) in an image are transparent.

XAspectRatio and YAspectRatio define the pixel aspect ratio of the image data. The aspect ratio is calculated by dividing XAspectRatio by YAspectRatio.

PageWidth and PageHeight describe the required resolution to display the image. If the image data were to be displayed at a resolution of 320×200, the values of these fields would be 320 and 200, respectively.

Color Map (CMAP) chunk

The optional CMAP chunk stores color information for the image data. CMAP data is actually an array of the following data type:

```
typedef struct _ColorMapEntry
{
    BYTE Red;           /* Red color component (0-255) */
    BYTE Green;         /* Green color component (0-255) */
    BYTE Blue;          /* Blue color component (0-255) */
} COLORMAPENTRY;
```

Red, Green, and Blue store color component intensity values in the range 0 to 255 for a single color. A value of 255,255,255 is white, and 0,0,0 is black.

The CMAP chunk contains an array of COLORMAPENTRYs as data. The number of elements in this array will vary depending upon the number of colors in the image data. The CMAP chunk structure follows:

```
typedef struct _ColorMapChunk
{
    char  ChunkId[4];           /* Chunk Identifier "CMAP" */
    DWORD Size;                 /* Size of chunk data in bytes */
    /* Chunk data starts here */
```


Ex.1007 / Page 504 of 1158
APPLE INC.

```
    COLORMAPENTRY Map[Size/3];  /* Color map data */
} COLORMAPCHUNK;
```

ChunkId contains the chunk content identifier "CMAP".

Size is the number of bytes in the Map field.

Map is the actual color map data. It is an array of COLORMAPENTRY values. There are typically $2^{BMHD.BitPlanes}$ entries in this array.

Body (BODY) chunk

The BODY chunk stores the actual image data as a BYTE array. The structure of the BODY chunk follows:

```
typedef struct _BodyChunk
{
    char  ChunkId[4];           /* Chunk Identifier "BODY" */
    DWORD Size;                 /* Size of chunk data in bytes */
    /* Chunk data starts here */
    BYTE  ImageData[];          /* Image data */
} BODYCHUNK;
```

ChunkId contains the chunk content identifier "BODY".

Size is the number of bytes in the ImageData field.

ImageData is the actual image data. Image data is an array of byte values and may be stored uncompressed, or compressed (using the IFF Packer encoding algorithm).

Pixel data, compressed or not, is always stored in separate bitplanes. Each scan line is made up of BMHD.BitPlanes rows of bytes. Each bitplane row encodes one bit from the pixel value. The pixel data appears as follows:

```
BODY
   scan-line 0
      bitplane 0                          pixel value's least significant bit
         byte 0                           bits for the leftmost eight pixels
         byte 1
           .
           .
           .
         byte RowBytes - 1                see RowBytes note below
      bitplane 1
        .
        .
        .
      bitplane BMHD.BitPlanes - 1
   scan-line 1
    .
```


Ex.1007 / Page 505 of 1158
APPLE INC.

```
    .
    .
  scan-line BMHD.Height - 1
```

RowBytes is the smallest even integer greater than BMHD.Width / 8, which can be found using:

```
RowBytes = ((BMHD.Width + 15) >> 4) << 1;
```

This is equivalent to saying that each bitplane row must contain an even number of bytes. Each bitplane row (scan line) is therefore word-aligned with padding before compression, if necessary.

If BMHD.Masking is 1, there will be an extra bitplane row in each scan line. In other words, each scan line will contain (BMHD.BitPlanes + 1) bitplane rows. This extra bitplane forms a 1-bit mask that is to be applied to the image when it is displayed. Depending on your intentions, you'll often discard the mask plane while decoding the BODY, but you have to remember to read the mask plane if it is there, because it's not included in BMHD.BitPlanes. BMHD.Masking = 1 is becoming less common, particularly on platforms other than the Amiga.

The pixel values in a BODY can be indexes into the palette contained in a CMAP chunk, or they can be literal RGB values. If there is no CMAP and if BMHD.BitPlanes is 24, the ILBM contains a 24-bit image, and the BODY encodes pixels as literal RGB values. The bitplanes for each scan line appear in the BODY in the following order:

```
scan-line 0
   red bit 0                          red least significant bit
   red bit 1
    .
    .
    .
   red bit 7
   green bit 0                        green least significant bit
    .
    .
    .
   green bit 7
   blue bit 0                         blue least significant bit
    .
    .
    .
   blue bit 7
scan-line 1
   red bit 0
    .
```


Ex.1007 / Page 506 of 1158
APPLE INC.

If the ILBM has no CMAP and if BMHD.BitPlanes is 8 (or occasionally, less than 8), the file contains a gray-scale image. Bitplanes are stored in the same least-to-most order of significance, and the full black-to-white range is assumed to be 0 to ($2^{BMHD.BitPlanes}$) – 1, and is usually 255. Note that if you're thinking about implementing an ILBM writer, you should include a CMAP with your gray-scale images, because most non-Amiga IFF readers will refuse to load an ILBM without a CMAP unless it is a 24-bit image.

ILBMs created for use on an Amiga may also contain pixel data in their BODY chunks that reflects display capabilities peculiar to the Amiga. Such Amiga-specific ILBMs must contain a CAMG (Commdore-AMiGa) chunk that identifies the Amiga display mode. CAMG data consists of a single DWORD which contains the viewmode value. The Amiga has built-in support for interpreting this value, but programs running on other platforms can safely test certain bits to identify Amiga-specific pixel data.

Hold-And-Modify (HAM) display mode

HAM (Hold-And-Modify) is a display mode that allows the Amiga to display 12-bit and 18-bit images using only 6 or 8 bits per pixel. HAM images can be identified by bit 11's being set to 1 in the CAMG chunk (CAMG mode & 0x0800 != 0).

The 8-bit HAM8 mode was introduced with the Amiga 4000 and Amiga 1200 models and provides a very good near-photographic-quality image display with only eight bitplanes of image data.

The color of any pixel in a HAM image may be any color from a standard 16-color palette, or the same as the color of the pixel to the immediate *left* with the top four bits of either the red, green, or blue components changed. On the left edge of the screen, the border color is used, which is color index zero from the 16-color palette.

HAM images store pixel values in the BODY chunk as codes that are divided into a mode in the high two bits and data in the remaining bits. Possible mode values are:

Ex.1007 / Page 507 of 1158
APPLE INC.

Mode	Meaning
00	Data bits are an index into the CMAP palette
01	Data bits are blue level
10	Data bits are red level
11	Data bits are green level

Unless a pixel is colormapped (mode 00), only one of its three RGB levels is given in its code. The other two are assumed to be the same as those for the pixel to its left. If the pixel is the first one in a scan line, the pixel to its left is assumed to be RGB(0, 0, 0).

The format of the mode and data bits in a pixel in HAM mode is:

5	4	3	2	1	0	
0	0	*w*	*x*	*y*	*z*	= Use colormap value *wxyz*
0	1	*w*	*x*	*y*	*z*	= Keep color from previous pixel, but change blue upper 4 bits to *wxyz*
1	0	*w*	*x*	*y*	*z*	= Keep color from previous pixel, but change red upper 4 bits to *wxyz*
1	1	*w*	*x*	*y*	*z*	= Keep color from previous pixel, but change green upper 4 bits to *wxyz*

A HAM image cannot be directly decoded into a standard 8-bit or lower palette-based image without further color reduction. For full quality, it must be converted to at least a 12-bit color image.

The number of data bits is 4 for standard HAM and 6 for HAM8, and the corresponding BMHD.BitPlanes value will normally be 6 or 8. The data bits should be precision-extended when the levels are decoded to 24-bits, and regardless of the number of data bits, the maximum level should translate to 255 at 8 bits per RGB channel.

The format of the mode and data bits in a pixel in HAM8 mode is:

7	6	5	4	3	2	1	0	
0	0	*n*	*m*	*w*	*x*	*y*	*z*	= Use color palette value *nmwxyz*
0	1	*n*	*m*	*w*	*x*	*y*	*z*	= Keep color from previous pixel, but change blue upper 6 bits to *nmwxyz*

Ex.1007 / Page 508 of 1158
APPLE INC.

7	6	5	4	3	2	1	0	
1	0	*n*	*m*	*w*	*x*	*y*	*z*	= Keep color from previous pixel, but change red upper 6 bits to *nmwxyz*
1	1	*n*	*m*	*w*	*x*	*y*	*z*	= Keep color from previous pixel, but change green upper 6 bits to *nmwxyz*

HAM8 images need to be directly converted to a 24-bit image in order to retain the full image quality on non-Amiga systems.

It is possible for the mode to be a single bit; BMHD.BitPlanes will then be either 5 or 7. The single bit is the low bit, while the high bit is assumed to be 0, implying that only the blue level can be modified. For obvious reasons, single mode bit images are rarely encountered.

Extra-Halfbrite (EHB) display mode

Extra-Halfbrite is another Amiga variant, now quite rare. The original Amiga models had a color palette of 32 colors (from a range of 4096) and could support up to 6 bitplanes. When 6 bitplanes were selected, one of two modes could be chosen to utilize the extra bit of data.

EHBs are 64-color (6 bitplane) images with 32-color palette entries. They can be identified by bit 7's being set to 1 in the CAMG (CAMG mode & 0x0080 != 0). Colors 32 to 63 are "half-bright" versions of colors 0 to 31.

To decode an EHB image, extend the color palette to 64 colors, and create colors 32 to 63 by copying and bit-shifting each palette color (0 to 31) right by one. The image is then decoded as it normally would be.

Image data compression

FORM ILBM files may contain image data compressed using a simple, run-length encoding algorithm called Packer. This algorithm is identical to the Macintosh PackBits algorithm and is also the algorithm used by the TIFF file format.

Packer encodes runs of identical byte values within a scan line. Encoding always stops at the end of every scan line. All byte runs are encoded as two-byte codes. The first byte is a code byte which indicates the type of compressed run and the number of pixels in the run. If the value of the code is 0 to 127 (signed bit off), the run is a literal run of pixels, and the next (code + 1) bytes are copied literally from the compressed data.

Ex.1007 / Page 509 of 1158
APPLE INC.

If the value of the code is –1 to –127 (signed bit on), the next byte following the code byte is read and its value is repeated (–*code* + 1) times. A code value of –128 is a no-op and is always ignored.

When compression is indicated, all of the image data stored in a BODY chunk is compressed, including any masking data interleaved with the image data.

For Further Information

For further information about the IFF format, see the following specifications on the CD-ROM that accompanies this book:

Morrison, Jerry, *EA IFF 85 Standard for Interchange Format Files*, Electronic Arts, 14 January 1985.

Morrison, Jerry, *ILBM IFF Interleaved Bitmap*, Electronic Arts, 17 January 1986.

These documents are also available from many online services and BBSs. You may also obtain them directly from the creator of IFF, Electronic Arts, at:

Electronic Arts
1820 Gateway Drive
San Mateo, CA 94404
Voice: 415-571-7171
Voice: 415-572-2787
WWW: *http://www.ea.com/*

The following documents from Electronic Arts are also widely available online:

FTXT: IFF Formatted Text, Electronic Arts. IFF supplement document for a text format.

ILBM: IFF Interleaved Bitmap, Electronic Arts. IFF supplement document for a raster image format.

Commodore-Amiga previously supported the IFF format, but the company was purchased by ESCOM, a German PC manufacturer.

The following Commodore-Amiga document is widely available online:

Scheppner, Carolyn, *Introduction to Amiga IFF ILBM Files and Amiga Viewmodes, Commodore Amiga Technical Support.*

Ex.1007 / Page 510 of 1158
APPLE INC.

The following books also discuss the IFF file format:

Commodore Amiga, Inc., *Amiga ROM KERNEL Reference Manual: Includes and Autodocs*, Addison-Wesley, Reading, MA, 1989.

Commodore Amiga, Inc., *Amiga ROM KERNEL Reference Manual: Devices*, Third Edition, Addison-Wesley, 1991.

The *RKM: Devices* manual contains more than 200 pages devoted to IFF and includes the complete text of every official Jerry Morrison specification and a generous amount of source code, including an implementation of Packer.

The Aminet archives are the best source of IFF information and display programs for the Amiga. Aminet mirrors include:

ftp://nic.funet.fi/pub/amiga/graphics/applications/convert
ftp://wuarchive.wustl.edu/pub/aminet/gfx/conv
http://wuarchive.wustl.edu/~aminet/dirs/gfx_conf.html

Inquiries about the Aminet archives may be emailed to *aminet.aminet.org*.

Ex.1007 / Page 511 of 1158
APPLE INC.

IGES

NAME:	IGES*
ALSO KNOWN AS:	Initial Graphics Exchange Specification
TYPE:	Vector and 3D
COLORS:	NA
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	NA
NUMERICAL FORMAT:	NA
ORIGINATOR:	IGES/PDES Organization
PLATFORM:	All
SUPPORTING APPLICATIONS:	Many high-end CAD packages
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Neutral file format for CAD-related data.

COMMENTS: IGES is used to share high-quality CAD data. It is not a proprietary format controlled by a private company. However, due to its complexity and difficulty, it is not commonly used by low-end CAD packages. Instead, it is most popular among expensive high-end CAD packages, such as those used in the automotive and aerospace industries.

Overview

IGES, the Initial Graphics Exchange Specification, is designed to exchange information between CAD systems and other vector-oriented applications. The standard is developed and maintained by the ANSI-accredited IGES/PDES Organization. The first version of IGES was adopted as American National Standard ANS Y14.26M-1981. Versions 3, 4 and the most recent version 5.2 were approved by ANSI as well.

* Our thanks to John Foust for his contributions to this article.

Ex.1007 / Page 512 of 1158
APPLE INC.

Like other CAD formats, such as AutoCAD DXF, an IGES file can represent many different types of data, ranging from lines and arcs to the complex geometric solids, such as cylinders and cones known as "constructive solid geometry." IGES is much more complicated than DXF, though—perhaps four times as complex, judging by the sheer number of different geometrical entities. The IGES v4 specification encompasses more than 500 pages.

Because of the format's complexity, it is difficult to implement every possible operation and entity. Most IGES translators or IGES-importing programs describe exactly which IGES entities they support and which they ignore.

IGES has both an ASCII and a binary format. The ASCII format is line-oriented, because of its early origins on punched card systems. There is also a compressed ASCII format that eliminates some of the punch-card anachronisms. The binary format is much more compact than the ASCII version.

IGES is associated with NCGA (National Computer Graphics Association) as part of the U.S. Product Data Association (USPRO) and the IGES/PDES Organization (IGO). NCGA administers the National IGES User Group (NIUG), which provides access to information and is a place to exchange information on IGES.

For Further Information

For more information about NIUG and obtaining the IGES file format specification from NIUG, contact:

National Computer Graphics Association
2722 Merrilee Drive
Suite 3200
Fairfax, VA 22031
Voice: 703-698-9600

NCGA can also give you information about the National Institute for Standards and Technology (NIST) testing for IGES.

On the World Wide Web, see the NIST's pages at:

http://www.eeel.nist.gov/iges/

Most serious CAD packages provide tools for working with IGES data. For example, see the pages at:

Ex.1007 / Page 513 of 1158
APPLE INC.

http://www.intergraph.com
http://www.autodesk.com

and search their indexes for "IGES."

Ex.1007 / Page 514 of 1158
APPLE INC.

NAME:	Inset PIX
ALSO KNOWN AS:	PIX
TYPE:	Bitmap
COLORS:	Up to 16
COMPRESSION:	Proprietary, documented
MAXIMUM IMAGE SIZE:	64K×64K pixels
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Inset Systems
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	InSet Versions 1 and 2, HiJaak, WordStar, Multimate
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Neutral common format for Inset Systems' products. Also used for graphics storage by the WordStar and Multimate word processors.

COMMENTS: A great little format marred by lack of support for more than 16 colors. It would be a good model for a deep-pixel format, however.

Overview

Inset PIX is an intermediate graphic format created by Inset Systems (now part of Quarterdeck), which sells the InSet and HiJaak applications for use on Intel-based PCs. HiJaak is a widely used and highly regarded screen-capture and graphics file conversion utility. Version 2 of the package supports the graphics, printer, and FAX formats listed below:

ASCII	Inset IGF
AT&T Group 4	Kofax Group 4
AutoCAD DXF	Lotus Picture
CALS raster	Macintosh Paint
CGM	Macintosh PICT

Ex.1007 / Page 515 of 1158
APPLE INC.

GIF	MathCAD
DataBeam DBX	Microsoft Paint
Dr. Halo	Microsoft Windows Bitmap
Encapsulated PostScript	Microsoft Windows Metafile
FAX formats (various)	PCX
GEM raster	Presentation Manager Metafile
GEM VDI	Tektronix P10
HPPCL	TGA
HPGL	TIFF
ILBM	WordPerfect Graphics Metafile
Inset PIX	

As you might imagine, Inset has a great deal of experience with graphics file formats. The design of PIX reflects this experience, and is a well thought-out and flexible format. If you need to convert a file from an odd format to one on the above list, you might consider converting to PIX as an intermediate step and then using the application to do the rest. (And, no, we don't have any financial or personal interest in Inset Systems or Quarterdeck!)

PIX was designed as an extensible, device-independent format which would allow random access of portions of a compressed image. Although nominally a bitmap format, the file structure supports the future addition of other data types.

File Organization

Inset documentation describes a PIX file as, "an indexed database of data items." A table of the size and location of data items is included in the beginning of each file. Data items can include information on the following:

- Image attributes, including dimensions, type, origin, and colors
- Information relevant at print time, such as clipping, size, and rotation
- Palette information
- Compressed sections of bitmap data ("image tiles")
- Bitmap tile-sectioning information

Ex.1007 / Page 516 of 1158
APPLE INC.

File Details

This section describes the header, index table, and different types of image data in a PIX file.

Header

PIX files always have a short header:

```
typedef struct _PIX_HEADER
{
  WORD RevisionLevel;      /* Currently 3 */
  WORD DataItemsInTable;   /* Number of data items in the index table */

} PIX_HEADER;
```

RevisionLevel is the version number for the format; this level is currently 3.

DataItemsInTable is the number of items in the PIX file's index table, described in the next section.

Index Table

Following the header is an index table containing an array of data item information structures of the following form:

```
typedef struct DATA_ITEM_INFO
{
  WORD DataID;          /* Data item ID */
  WORD DataLength;      /* Length of data item */
  LONG DataLocation;    /* Location of data item in file */

} DATA_ITEM_INFO;
```

DataID values may be any of the following:

00 Image information
11 Printing options
01 Palette
02 Tile information

Empty data items have DataID values of -1.

DataLength is the length of this item in the index table.

DataLocation is the location of the item in the file.

Ex.1007 / Page 517 of 1158
APPLE INC.

Image Information

The following information is extracted from the specification document supplied by Inset Systems:

The application ID for the image structure is 0. This data item contains information on the overall image size, type, and origin of the image. The structure (in C) of this data item is:

```
struct mode_data
{
  BYTE    hmode;
  BYTE    htype;
  BYTE    cfore;
  BYTE    cback;
  BYTE    tattr;
  BYTE    tcpr;
  BYTE    trows;
  BYTE    thfnts;
  BYTE    tlfnts;
  BYTE    tcpf;
  BYTE    tfsize[4];
  WORD    gcols;
  WORD    grows;
  BYTE    gfore;
  BYTE    prepal;
  BYTE    lodpal;
  BYTE    lintens;
  BYTE    lred;
  BYTE    lgreen;
  BYTE    lblue;
  BYTE    pages;
  BYTE    haspect;
  BYTE    vaspect;
}
```

Following is a description of each member of this structure:

hmode — Hardware-specific mode. 0 if not specifically related to a particular hardware mode of a board.

htype — Hardware type. Bit 0 is zero if alphanumeric and 1 if bitmap graphics. Board types are ORed into this byte. Board types include:

8 = CGA
16 = Hercules

24 = EGA

Ex.1007 / Page 518 of 1158
APPLE INC.

cfore	Text foreground color bits (ignore if graphics) 4 for CGA
tattr	Text background color bits (ignore if graphics) 3 for CGA
tcpr	Text characters per row (ignore if graphics)
trows	Text rows (ignore if graphics)
thfnts	Text hardware fonts (not used)
tlfnts	Text loadable fonts (not used)
tcpf	Text characters per font (not used)
tfsize	Font size (not used)
gcols	Graphics columns
grows	Graphics rows
gfore	Graphics foreground color bits
prepal	Number of preset palettes (default to 0)
lodpal	Number of loadable palettes
lintens	Number of palette bits for intensity
lred	Number of palette bits for red
lgreen	Number of palette bits for green
lblue	Number of palette bits for blue
pages	Number of possible pages (not used)
haspect	Horizontal component of aspect ratio (number of horizontal pixels to fit in a square)
vaspect	Vertical component of aspect ratio (number of vertical pixels to fit in a square)

A sample image structure for a 600-row-by-800-column single-bit plane image might be initialized as follows:

```
struct mode_data
{
  BYTE   hmode;      /* 0 */
  BYTE   htype;      /* 1 */
  BYTE   cfore;      /* 0 */
  BYTE   cback;      /* 0 */
  BYTE   tattr;      /* 0 */
  BYTE   tcpr;       /* 80 */
  BYTE   trows;      /* 25 */
```


Ex.1007 / Page 519 of 1158
APPLE INC.

```
    BYTE    thfnts;     /* 0 */
    BYTE    tlfnts;     /* 0 */
    BYTE    tcpf;       /* 0 */
    WORD    tfsize[4];  /* 0,0,0,0 */
    WORD    gcols;      /* 800 */
    WORD    grows;      /* 500 */
    BYTE    gfore;      /* 1 */
    BYTE    prepal;     /* 0 */
    BYTE    lodpal;     /* 0 */
    BYTE    lintens;    /* 1 */
    BYTE    lred;       /* 0 */
    BYTE    lgreen;     /* 0 */
    BYTE    lblue;      /* 0 */
    BYTE    pages;      /* 0 */
    BYTE    haspect;    /* 1 */
    BYTE    vaspect;    /* 1 */
}
```

Printing Options

The application ID for Inset Printing options is 11h. The C structure containing these items follows:

```
struct prt_options
{
  SHORT pitch;
  SHORT scol;
  SHORT ecol;
  SHORT srow;
  SHORT erow;
  SHORT p_wid;
  SHORT siz;
  SHORT rotat;
  SHORT do_sw;
  SHORT res_1;
  SHORT res_2;
  SHORT pcolor;
  SHORT row_dp;
  SHORT col_dp;
  SHORT flags;
  CHAR  ink_tab[16];
}
```

Following is a description of each member of this structure:

pitch — Printer Pitch divided by 10 (e.g., 100 = 10 pitch); not required, set to 100

Ex.1007 / Page 520 of 1158
APPLE INC.

scol	Start column clip boundary
ecol	End column clip boundary
srow	Start row clip boundary
erow	End row clip boundary
p_wid	Printer width (not required, set to 0)
size	Size (not used, set to 0)
rotat	Rotation (0 = horizontal, -1 = left, 1=right)
do_sw	Option bits ORed. Applicable bits to set include: Double Pass | = 2 Letter Quality | = 4 Border On | = 10 (hex)
res_1	
res_2	Internal use (don't use, set to 0)
pcolor	Low-order byte indicates whether the image settings are intended for color printer (true = 1, false = 0) High-order byte indicates which dither pattern type to use (0 = old, 1 = gray, 2 = contrast).
row_dp	Height of image in decipoints (1/720 inches)
col_dp	Width of image in decipoints (1/720 inches)
flags	Modify flags Size option bits: #define INDATA_USE_COL 0x1 #define INDATA_USE_INCH 0x2 #define INDATA_USE_DP 0x3 Modify ink selection: #define INDATA_INK_INV 0x8 #define INDATA_INK_BW 0x10 #define INDATA_INK_TAB 0x18

Ex.1007 / Page 521 of 1158
APPLE INC.

```
#define INDATA_VDISK   0x20
#define INDATA_VSIZE   0x40
#define INDATA_DYNAMIC 0x80
```

ink_tab Sixteen-byte table mapping screen colors to printer colors/gray patterns (see the MODIFY/INKS section of the Inset manual for more information on ink tables). If you want to select your own Ink table mapping, the flags variable must have a 0x18 OR'ed in. Preset Inset Ink tables are set as follows:

```
Color Number      0 1 2 3 4 5 6 7 8 9 A B C D E F

Standard                F 1 2 3 4 5 6 7 8 9 A B C D E 0
Invert            0 1 2 3 4 5 6 7 8 9 A B C D E F
B&W                     F 0 0 0 0 0 0 0 F 0 0 0 0 0 0 0
```

Palette Data

Application ID = 1 contains display palette information. Palette information is stored in an array of palette structures of the following form:

```
struct pallette
{
  CHAR intens, red, green, blue;
}
```

The number of significant palette items in the array is determined by the number of available colors (the "gfore" member in the image data structure) in the image. The significant bits are determined by the lintens lred, lgreen, and lblue items in the mode_data structure.

Tile Information

Application ID = 2 contains information as to how the image is broken down into tiles. The Tile_Data structure follows:

```
struct Tile_Data
{
  WORD    page_rows;
  WORD    page_cols;
  WORD    stp_rows;
  WORD    stp_cols;
}
```


Ex.1007 / Page 522 of 1158
APPLE INC.

Following is a description of each member of this structure:

page_rows	Number of rows within each tile
page_cols	Number of columns in each tile (must be divisible by 8)
stp_rows	Number of horizontal tile strips within the image
stp_cols	Number of vertical tile strips

Each tile is limited to a maximum of 4096 bytes of uncompressed data. The actual tiles are numbered starting with the upper left row as tile 0 and incremented from left to right as illustrated below.

0	1	2	3
4	5	6	7
8	9	A	B

The ID of a tile is the tile number ORed with a 8000h. For example, the lower-right tile can be found by finding the record with ID=800Bh.

Images may be broken down into checkerboard sections or horizontal strips. However, if an image is broken into horizontal strips for processing, the image as rotated by Inset will be slowed.

Pixel Tiles

Each individual pixel tile ID is determined by ORing in 8000h with the tile number, as described in the preceding section on tile information.

The image is organized into bit planes with eight pixels per byte; the most significant bit contains the leftmost pixel. For multiple-bit plane images, all scan lines for a plane are written out before the scan lines in the next plane.

If the actual column boundary of the tile exceeds the column boundary of the image, the image is padded with blank bytes to fill out the tile. If the actual row boundary of the tile exceeds that of the image, the extra rows are not present.

When the tile is stored on disk, it is in a vertically compressed format. The first scan line of each tile is written out with no modification. Then, before the following scan lines, there are compression bytes that indicate which bytes in the scan line are different from the preceding line. Each bit in the compression byte indicates whether a particular byte in that scan has changed (1 if changed, 0 if not). Then, following the compression bytes, only the modified scan bytes are written to the file.

Ex.1007 / Page 523 of 1158
APPLE INC.

For example, suppose we have a tile that is eight columns wide, the first scan line is all blank, and there is a dot at the beginning and end of the second scan line. This tile would be written to disk as follows:

```
   00 00 00 00 00 00 00 00 ← First scan line
81 80                   01 ← changed bytes
 ↑
 Compression Byte
```

In multiple-bit plane images, the first scan is uncompressed, and the following lines are compressed in the same manner as described above.

Character Tiles

Alphanumeric images can be generally described as two-plane images with the first plane containing the alphanumeric character data and the second containing the attribute information. Alphanumeric characters are presumed to correspond to the IBM extended ASCII character set, with attribute information corresponding to the IBM CGA standard.

The character and text planes are compressed in the same manner as image bit planes with one caveat. The text scan-line length is twice what it should be (i.e., 160 bytes go out uncompressed for an 80-column screen).

For Further Information

For further information about Inset PIX, see the specification included on the CD-ROM that accompanies this book. You can also contact:

Inset Systems
Developer Relations
71 Commerce Drive
Brookfield, CT 06804
Voice: 203-740-2400

Also see the Quarterdeck homepage:

http://www.insetusa.com/

Ex.1007 / Page 524 of 1158
APPLE INC.

NAME:	Intel DVI
ALSO KNOWN AS:	DVI, Digital Video Interface, Intel Real-Time Video
TYPE:	Multimedia
COLORS:	16 million
COMPRESSION:	JPEG, proprietary
MAXIMUM IMAGE SIZE:	256×240
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	NA
ORIGINATOR:	Intel Corporation
PLATFORM:	Intel-based PCs
SUPPORTING APPLICATIONS:	MS-DOS and Microsoft Windows Multimedia
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	Microsoft RIFF, QuickTime

USAGE: A format designed to support Intel's bid to establish a hardware standard in the Intel-based PC marketplace.

COMMENTS: At the time of this writing, it's too early to decide whether this format will ever see the light of day in a commercial product. If it ever does, you'll find the specification on the CD-ROM useful.

Overview

Intel is the current owner of DVI, which was one of the first systems that provided practical full-motion video incorporating real-time decompression technology. DVI originated in 1984 at the David Sarnoff Research Center in Princeton, New Jersey, which was the central research facility for RCA Corporation. Ownership of DVI changed in 1986 when RCA was acquired by GE. The official unveiling of GE DVI occurred in March 1987 at the Second Microsoft CD-ROM Conference in Seattle, Washington. GE later sold DVI technology to Intel Corporation in October 1988.

DVI is actually both the name of the Digital Video Interactive hardware system sold by Intel and the file format associated with that system. DVI technology is essentially a PC-based interactive audio/video system used for multimedia

Ex.1007 / Page 525 of 1158
APPLE INC.

applications. The DVI system consists of a board for use in an Intel-based PC, drivers, and associated software. The four components of DVI technology are:

- DVI hardware chipset
- Run-time software interface
- Data compression and decompression schemes
- Data file formats

The heart of the DVI system is the hardware architecture based on the video display processor (VDP) chipset. DVI technology was originally designed for implementation on the IBM PC AT platform. A collection of three 16-bit, ISA-bus DVI interface boards (audio, video, and CD-ROM) were plugged into the AT, and all of the hardware capabilities were accessed through the run-time software interface. The functions in the interface were called by writing a software program using a programming language such as assembly or C.

Today, Intel distributes licenses to third-party developers who want to incorporate DVI technology into their platforms and multimedia products. All of IBM's multimedia hardware platforms (such as the ActionMedia II boards) and software applications are based upon DVI technology.

DVI is a major competitor of QuickTime, AVI, and MPEG for market share in digital audio/video applications.

DVI allows the storage and playback of audio and video information. All DVI images have a 5:4 pixel aspect ratio and are 256×240 pixels in size. DVI is also capable of storing still images and supports both a lossy and a lossless native compression method for such images. DVI works across MS-DOS, Microsoft Windows, and OS/2 platforms and supports the capability of using its own proprietary compression scheme, or using user-definable algorithms, such as JPEG, as well. Audio compression is achieved using either the ADPCM or PCM8 algorithms.

File Organization

The DVI file format is extremely flexible in its design and is used to store a wide variety of data. This format is capable of storing both still-image and motion-video/audio data. The type of data a DVI file contains is described by its file extension. The common extensions for DVI files containing still-image data are the following:

Ex.1007 / Page 526 of 1158
APPLE INC.

Uncompressed, 8-bit data

.IMR	Red channel
.IMG	Green channel
.IMB	Blue channel
.IMY	Y luminance channel
.IMI	I color channel
.IMQ	Q color channel
.IMM	Monochrome or gray scale
.IMA	Alpha channel
.IMC	Color map
.I8	Device-dependent data

Uncompressed, 16-bit data

.I16	Device-dependent data

Compressed, 8-bit data

.CMY	Y luminance channel
.CMI	I color channel
.CMQ	Q color channel

Compressed, 16-bit data

.C16	Device-dependent data

As you can see, a common practice of DVI is to store each color plane of an image in a separate disk file. This allows the easy reading and writing of bitmap information, without the need to buffer data to read or write a single file.

A still image is saved using three color-channel files and possibly a colormap and alpha-channel file as well. Motion-video/audio data is stored using the Audio/Video Support System (AVSS) file format. AVSS (pronounced "avis") allows audio and video data to be stored in the same file and played back in a synchronized manner. All AVSS files have the extension .AVS or the file type AVSS.

File Details

The data in AVSS files is primarily stream-based, and there is at least one data stream per AVSS file. Each file contains a standard header, an AVL file header,

Ex.1007 / Page 527 of 1158
APPLE INC.

one stream header per data stream, one substream header per substream, frame data, and a frame directory.

DVI File Header

The standard header of a DVI file is 12 bytes long and has the following structure:

```
typedef _DviStandardHeader
{
  DWORD FileId;           /* Magic number (56445649h) of DVI file */
  SHORT HeaderSize;       /* Size of this header structure */
  SHORT HeaderVersion;    /* Version of this header structure */
  DWORD AnnotationOffset; /* Location of annotation data */

} DVISTANDARDHEADER;
```

FileId contains the characters VDVI and identifies the file as containing DVI information. If the file contains still-image information, this field contains the characters VIM.

HeaderSize contains the number of bytes found in the header, which is currently 12. Older versions of the DVI format may contain a value of 1 in this field. In this case, this value should be ignored and treated as if it were 12.

HeaderVersion indicates the format of the header and is currently 1.

AnnotationOffset is used to point at additional, unstructured data, such as a title or copyright notice, which is normally placed at the end of the file. If no annotation exists, then this field is set to 0.

AVL File Header

The AVL file header immediately follows the standard header and is a directory of all the other data structures within the DVI file. This header is 120 bytes in length and has the following format:

```
typedef struct _AvlHeader
{
  DWORD HeaderId;              /* Header ID value (41565353h) */
  SHORT HeaderSize;            /* Size of this header structure */
  SHORT HeaderVersion;         /* Format of this header structure */
  SHORT StreamGroupCount;      /* Number of stream groups in the file */
  SHORT StreamGroupSize;       /* Size of each stream group */
  DWORD StreamGroupOffset;     /* Location of the first stream group */
  SHORT StreamGroupVersion;    /* Format of each stream group */
  SHORT StreamSize;            /* Size of the stream header */
  SHORT StreamVersion;         /* Format of the stream header */
```


Ex.1007 / Page 528 of 1158
APPLE INC.

```
    SHORT StreamCount;           /* Number of stream headers in the file */
    DWORD StreamOffset;          /* Location of stream structures array */
    DWORD HeaderPoolOffset;      /* Location of substream headers */
    LONG  LabelCount;            /* Number of labels in the file */
    DWORD LabelOffset;           /* Location of the first label */
    SHORT LabelSize;             /* Size of each label */
    SHORT LabelVersion;          /* Format of each label */
    DWORD VideoSeqHeaderOffset;  /* Location of video sequence header */
    WORD  VideoSeqHeaderSize;    /* Size of video sequence header */
    SHORT FrameVersion;          /* Version of frame headers in file */
    LONG  FrameCount;            /* Number of frame headers in file */
    LONG  FrameSize;             /* Size of frame header and data */
    DWORD FirstFrameOffset;      /* Location of the first frame */
    DWORD EndOfFrameOffset;      /* Location of last frame byte + 1 */
    SHORT FrameHeaderSize;       /* Size of frame header */
    SHORT FrameDirectorySize;    /* Size of the frame directory */
    DWORD FrameDirectoryOffset;  /* Location of the frame directory */
    SHORT FrameDirectoryVersion; /* Format of the frame directory */
    SHORT FramesPerSecond;       /* Frame rate of the data */
    DWORD UpdateFlag;            /* Data is updating or complete */
    DWORD FreeBlockOffset;       /* Not used */
    BYTE  Patch[32];             /* Not used */

} AVLHEADER;
```

HeaderId contains the characters AVSS and identifies the header as containing AVL file information.

HeaderSize contains the number of bytes found in the header. This value is currently 120.

HeaderVersion contains a value that specifies the format of the header based on a version control rating. Each modification to the header structure increments the header version. The current HeaderVersion value for the AVLHEADER structure is 3.

If the streams within a DVI file are organized as groups, then the StreamGroupCount value indicates the number of groups; the StreamGroupSize value specifies the size of each group; the StreamGroupOffset value points to the location of the first group; and the StreamGroupVersion specifies the format of the group. If no stream groups are present in the file, then the value of these fields will be 00h.

The next four fields contain information on the array of STREAMHEADER structures stored in the DVI file. StreamSize indicates the size of each structure, which is currently 44. StreamVersion specifies the format of each structure,

Ex.1007 / Page 529 of 1158
APPLE INC.

which is currently a value of 3. StreamCount is the number of streams in the file and structures in the array. StreamOffset contains the offset value to the beginning of the array.

HeaderPoolOffset points to the first substream header. This value is 00h if there are no substreams present in the file.

If the DVI file contains labels, then LabelCount indicates the number of labels; LabelOffset points to the location of the first label; LabelSize specifies the size of each label; and LabelVersion specifies the format of the label. If no labels are present in the file, then the values of these fields are all 00h.

VideoSeqHeaderOffset and VideoSeqHeaderSize describe the location and size of the video sequence header, if one is present in the file.

FrameVersion indicates the format of the data frames and is currently 3. FrameCount is the number of frames in the file, and FrameSize is the size of a frame, including its header. FirstFrameOffset is the location of the first frame header.

EndOfFrameOffset points to the location of the first byte after the frame data.

FrameHeaderSize value is the size of the frame header.

The FrameDirectorySize specifies the size of the frame directory and is currently 4.

FrameDirectoryOffset points to the location of the frame directory.

FrameDirectoryVersion specifies the format of the frame directory.

FramesPerSecond contains the frame rate of the data for playback, rounded to the nearest integer.

UpdateFlag is a non-zero value if the file is in the process of being updated. A value of 00h indicates that the file is not currently being modified.

FreeBlockOffset and Patch[32] are set to 00h.

Stream Header

Each DVI file contains one or more data streams. Each stream is identified by an associated STREAMHEADER structure, which contains detailed information about the stream data. This header is four bytes in length and has the following format:

Ex.1007 / Page 530 of 1158
APPLE INC.

```
typedef struct _StreamHeader
{
DWORD HeaderId;            /* Header ID value (5354524Dh) */
SHORT HeaderSize;          /* Size of this header structure */
WORD  Type;                /* The type of data stream */
WORD  SubType;             /* The subtype of data stream  */
SHORT HeaderCount;         /* Number of substream headers */
SHORT NextStreamNumber;    /* ID of the next stream */
SHORT StreamGroupNumber;   /* The group ID for this stream */
SHORT Pad;                 /* Pad value */
DWORD Flag;                /* Variable frame size flag */
LONG  FrameSize;           /* Maximum amount of data per frame */
DWORD FirstHeaderOffset;   /* Location of first substream header */
BYTE  StreamName[16];      /* Name of the stream */

} STREAMHEADER;
```

HeaderId contains the characters STRM and identifies the header as containing AVL file information.

HeaderSize contains the number of bytes found in the header. This value is currently 120.

Type and SubType indicate the type of data that is stored in this stream. Valid Type values are:

02h Compressed audio stream
03h Compressed image stream
05h Associated per-frame data
06h Uncompressed image stream
07h Pad stream

SubType indicates a variation of each of these stream types and is described for different types of streams (e.g., video) in the following sections.

HeaderCount indicates the number of substreams associated with this stream.

NextStreamNumber is not used and is set to –1.

StreamGroupNumber indicates the ID of the group this stream is associated with.

Pad is not used and is set to 00h.

Flag is 04h if the stream contains frames that vary in size; otherwise, the value will be 00h.

FrameSize field specifies the maximum number of bytes per frame in the stream.

Ex.1007 / Page 531 of 1158
APPLE INC.

FirstHeaderOffset points to the location of the first substream header.

StreamName is the name of the stream in the form of a NULL-terminated ASCII string.

Audio substream header

Each type of data stream has a substream header. The audio substream header describes the attributes of an audio stream. The type of audio stream is indicated by the SubType field in the STREAMHEADER structure. For audio streams, this value is always 0. The AUDIOSUBSTREAMHEADER header is 168 bytes in length and is formatted as follows:

```
typedef struct _AudioSubStreamHeader
{
  DWORD HeaderId;          /* Header ID value (41554449h) */
  SHORT HeaderSize;        /* Size of this header structure */
  SHORT HeaderVersion;     /* Format of this header structure */
  BYTE  OriginalFile[80];  /* Name of file stream is derived from */
  LONG  OriginalFrame;     /* Original frame ID */
  SHORT OriginalStream;    /* Original stream ID */
  SHORT Pad;               /* Pad value */
  LONG  FrameCount;        /* The number of frames */
  DWORD NextHeaderOffset;  /* Location of next substream header */
  BYTE  LibraryName[16];   /* Name of library stream if from */
  BYTE  AlgorithmName[16]; /* Audio compression algorithm used */
  LONG  DataRate;          /* Audio data rate in bits/sec */
  SHORT CutoffFrequency;   /* Filter cutoff frequency */
  SHORT Parameter3;        /* Not used */
  SHORT LeftVolum rate;    /* Loudness of left audio channel */
  SHORT RightVolume;       /* Loudness of right audio channel */
  LONG  LoopOffset;        /* Not used */
  LONG  StartingFrame;     /* ID of the first frame in the stream */
  DWORD Flag;              /* Mono/Stereo flag */
  SHORT FrameRate;         /* The playback rate for this stream */
  SHORT Pad2               /* Pad value */
  LONG  DCFId;             /* Digital Compression Facility ID */

} AUDIOSUBSTREAMHEADER;
```

HeaderId contains the characters AUDI and identifies the header as containing audio stream information.

HeaderSize contains the number of bytes found in this header. This value is currently 168.

HeaderVersion indicates the version number of the header. The current HeaderVersion field value is 5.

Ex.1007 / Page 532 of 1158
APPLE INC.

OriginalFile contains a NULL-terminated ASCII string identifying the name and path of the file, from which the audio information is derived.

OriginalFrame, OriginalStream, and Pad are not used and are set to 0.

FrameCount indicates the number of frames in the audio stream.

NextHeaderOffset specifies the location of the next audio substream header. This value is always 00h.

LibraryName is not used and should be set to all NULL values.

AlgorithmName contains a NULL-terminated ASCII string that identifies the name of the compression method used on the audio data stream. This string is apdcm4e or pcm8 for the ADPCM and PCM8 algorithms, respectively.

DataRate is the data rate of the audio stream in bits per second.

CutoffFrequency indicates the maximum filter cutoff frequency for the sample.

Parameter3 is not used and is set to 0.

LeftVolume and RightVolume specify the volume level of the left and right audio channels, respectively. These numbers are a percentage of the total volume; the default value is 100.

LoopOffset has a default value of –1.

StartingFrame normally has a value of 0.

Flag has a value of 4000h, indicating that the audio stream is stereophonic, or 8000h, indicating that it is monophonic.

FrameRate is the data rate of the audio stream.

Pad2 is not used and is set to 0.

DCFId contains a value indicating the software service that compressed the data. The value of this field is –1 if the ID of the service is not known or is unimportant.

Video substream header

The video substream header describes the attributes of a video or compressed image stream. The type of video stream is indicated by the SubType field in the STREAMHEADER structure, which may have the following values for a video stream:

Ex.1007 / Page 533 of 1158
APPLE INC.

1 Y-channel data only
11 U-channel data only
12 V-channel data only
13 YVU data
14 YUV data

All images are stored in YVU format, except for JPEG-compressed images, which are stored using YUV.

The VIDEOSUBSTREAMHEADER header is 136 bytes in length and is formatted as follows:

```
typedef struct _VideoSubStreamHeader
{
  DWORD HeaderId;          /* Header ID value (h) */
  SHORT HeaderSize;        /* Size of this header structure */
  SHORT HeaderVersion;     /* Format of this header structure */
  BYTE  OriginalFile[80]; /* Name of file stream is derived from */
  LONG  OriginalFrame;     /* Original frame ID */
  SHORT OriginalStream;    /* Original stream ID */
  SHORT Pad;               /* Pad value */
  LONG  FrameCount;        /* Number of frames until next header */
  DWORD NextHeaderOffset; /* Location of next substream header */
  SHORT XPosition;         /* X coordinate top-left corner of image */
  SHORT YPosition;         /* Y coordinate top-left corner of image */
  SHORT XLength;           /* Width of image */
  SHORT YLength;           /* Height of image */
  SHORT XCrop;             /* X cropping coordinate */
  SHORT YCrop;             /* Y cropping coordinate */
  SHORT DropFrame;         /* Not used */
  SHORT DropPhrase;        /* Not used */
  LONG  StillPeriod;       /* Frequency of intraframe images */
  SHORT BufferMinimum;     /* Minimum buffer size required */
  SHORT BufferMaximum;     /* Maximum buffer size required */
  SHORT DecodeAlgorithm;   /* ID of the decompression algorithm */
  SHORT Pad2;              /* Pad value */
  LONG  DCFId;             /* Digital Compression Facility ID */

} VIDEOSTREAMHEADER;
```

HeaderId contains the characters CMIG and identifies the header as containing audio stream information.

HeaderSize contains the number of bytes found in this header. This value is currently 136.

HeaderVersion currently has a value of 4.

Ex.1007 / Page 534 of 1158
APPLE INC.

OriginalFile contains a NULL-terminated ASCII string identifying the name and path of the file from which the video or image information is derived.

OriginalFrame, OriginalStream, and Pad are not used and are set to 0.

FrameCount indicates the number of frames in the current substream.

NextHeaderOffset specifies the location of the next video substream header. This field value is always 00h.

XPosition and YPosition indicate the position of the top-left corner of the image. These fields are normally 0.

XLength and YLength specify the maximum width and height of the images stored in this stream.

XCrop and YCrop specify alternate length values used to crop the image. These values are 0 by default.

DropFrame and DropPhrase are not used and are set to 0.

StillPeriod indicates the interval at which intraframe encoding occurs. For example, a value of 12 in this field indicates that every 12th frame in this video stream is intraframe encoded. A value of 1 indicates that every frame is intraframe encoded. The default value of –1 indicates that the intraframe interval is unknown.

BufferMinimum and BufferMaximum indicate the extremes of the buffer sizes required for decompressing the image. These fields are normally set to 00h.

DecodeAlgorithm contains a value identifying the algorithm, needed to decompress the stream. Pad2 is not used and is set to 0.

DCFId contains a value indicating the software service that compressed the data. The value of this field is –1 if the ID of the service is unknown or not important.

For information on other substream header formats, refer to the DVI specifications on the CD-ROM.

Frames

Each section of frame data in a DVI file is preceded by a header describing the data in the frame. The structure of this header is shown below.

```
typedef struct _FrameHeader
{
  LONG FrameNumber;        /* Sequence number of this frame */
  LONG PreviousOffset;     /* Location of previous frame */
```


Ex.1007 / Page 535 of 1158
APPLE INC.

```
    LONG Checksum;          /* Checksum value for this frame */
    LONG StreamFrameSize[]; /* Array of all frame sizes */

} FRAMEHEADER;
```

FrameNumber stores the sequence number of the frame. PreviousOffset points to the location of the previous frame. This value is 00h if it is the first frame.

Checksum contains a checksum value of the frame header.

StreamFrameSize is an array of byte count values, one for each frame stored in the stream.

The location of each frame is stored in a directory of offset values. There will be one FRAMEDIRECTORY structure per frame stored in the stream. The format of this structure is shown below.

```
typedef struct _FrameDirectory
{
DWORD FrameOffset;      /* Location of the frame for this directory */
} FRAMEDIRECTORY;
```

FrameOffset points to the location of its associated frame. If the most significant bit of this value is set to 1, this offset may be used for access to the frame data of every stream in the file. Typically, only audio streams are suitable for random access.

For Further Information

For further information about the Intel DVI format, see the specification included on the CD-ROM that accompanies this book. The specifications for DVI and the AVSS file format may also be found in the reference material from the DVI Developer's Kit available from Intel:

Intel Corporation
Attn: Intel Action Media Support
6505 West Chandler Blvd
Chandler, AZ 85226
Voice: 602-554-4231
WWW: *http://www.intel.com/contents.html*

See the following books and articles for additional information about Intel DVI:

Dixon, D.F., S.J. Golkin, and I.H. Hashfield, "DVI Video Graphics," *Computer Graphics World*, July 1987.

Ex.1007 / Page 536 of 1158
APPLE INC.

Hurst, R.N., and A.C. Luther, "DVI: Digital Video from a CD-ROM," *Information Display*, April 1988.

Luther, Arch C. *Digital Video in the PC Environment: Featuring DVI Technology*, McGraw-Hill, New York, NY, 1989

Ex.1007 / Page 537 of 1158
APPLE INC.

JPEG File Interchange Format

NAME:	JPEG File Interchange Format
ALSO KNOWN AS:	JFIF, JFI, JPG, JPEG
TYPE:	Bitmap
COLORS:	Up to 24-bit
COMPRESSION:	JPEG
MAXIMUM IMAGE SIZE:	64K×64K pixels
NUMERICAL FORMAT:	Big-endian
MULTIPLE IMAGES PER FILE:	No
ORIGINATOR:	C-Cube Microsystems
PLATFORM:	All
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes (in JPEG package)
IMAGES ON CD:	Yes
SEE ALSO:	Chapter 9, *Data Compression* (JPEG section)

USAGE: Used primarily in graphics and image manipulation programs

COMMENTS: One of the few formats incorporating JPEG compression and as such offers superior compression for deep-pixel images.

Overview

JPEG (Joint Photographic Experts Group) refers to a standards organization, a method of file compression, and sometimes a file format. In fact, the JPEG specification itself, which we describe in terms of compression in Chapter 9, does not itself define a common file interchange format to store and transport JPEG data between computer platforms and operating systems. The JPEG File Interchange Format (JFIF) is a development of C-Cube Microsystems for the purpose of storing JPEG-encoded data. JFIF is designed to allow files containing JPEG-encoded data streams to be exchanged between otherwise incompatible systems and applications.

A JFIF file is basically a JPEG data stream with a few restrictions and an identfying marker. In order to understand the JFIF format, you'll need to understand

Ex.1007 / Page 538 of 1158
APPLE INC.

JPEG; in addition to Chapter 9, see the JPEG FAQ (Frequently Asked Questions) document included on the CD-ROM and available on the Internet.

File Organization

Both JPEG and JFIF data are byte streams, always storing 16-bit word values in big-endian format. JPEG data in general is stored as a stream of blocks, and each block is identified by a marker value.

The first two bytes of every JPEG stream are the Start Of Image (SOI) marker values FFh D8h. In a JFIF-compliant file there is a JFIF APP0 (Application) marker, immediately following the SOI, which consists of the marker code values FFh E0h and the characters JFIF in the marker data, as described in the next section. In addition to the JFIF marker segment, there may be one or more optional JFIF extension marker segments, followed by the actual image data.

File Details

Although JFIF files do not possess a formally-defined header, the SOI and JFIF APP0 markers taken together act as a header in the following marker segment structure:

```
typedef struct _JFIFHeader
{
  BYTE SOI[2];          /* 00h  Start of Image Marker     */
  BYTE APP0[2];         /* 02h  Application Use Marker    */
  BYTE Length[2];       /* 04h  Length of APP0 Field      */
  BYTE Identifier[5];   /* 06h  "JFIF" (zero terminated) Id String */
  BYTE Version[2];      /* 07h  JFIF Format Revision      */
  BYTE Units;           /* 09h  Units used for Resolution */
  BYTE Xdensity[2];     /* 0Ah  Horizontal Resolution     */
  BYTE Ydensity[2];     /* 0Ch  Vertical Resolution       */
  BYTE XThumbnail;      /* 0Eh  Horizontal Pixel Count    */
  BYTE YThumbnail;      /* 0Fh  Vertical Pixel Count      */

} JFIFHEAD;
```

SOI is the start of image marker and always contains the marker code values FFh D8h.

APP0 is the Application marker and always contains the marker code values FFh E0h.

Length is the size of the JFIF (APP0) marker segment, including the size of the Length field itself and any thumbnail data contained in the APP0 segment.

Ex.1007 / Page 539 of 1158
APPLE INC.

Because of this, the value of Length equals 16 + 3 * XThumbnail * YThumbnail.

Identifier contains the values 4Ah 46h 49h 46h 00h (JFIF) and is used to identify the code stream as conforming to the JFIF specification.

Version identifies the version of the JFIF specification, with the first byte containing the major revision number and the second byte containing the minor revision number. For version 1.02, the values of the Version field are 01h 02h; older files contain 01h 00h or 01h 01h.

Units, Xdensity, and Ydensity identify the unit of measurement used to describe the image resolution. Units may be 01h for dots per inch, 02h for dots per centimeter, or 00h for none (use measurement as pixel aspect ratio). Xdensity and Ydensity are the horizontal and vertical resolution of the image data, respectively. If the Units field value is 00h, the Xdensity and Ydensity fields will contain the pixel aspect ratio (Xdensity : Ydensity) rather than the image resolution. Because non-square pixels are discouraged for portability reasons, the Xdensity and Ydensity values normally equal 1 when the Units value is 0.

XThumbnail and YThumbnail give the dimensions of the thumbnail image included in the JFIF APP0 marker. If no thumbnail image is included in the marker, then these fields contain 0. A thumbnail image is a smaller representation of the image stored in the main JPEG data stream (some people call it an icon or preview image). The thumbnail data itself consists of an array of XThumbnail * YThumbnail pixel values, where each pixel value occupies three bytes and contains a 24-bit RGB value (stored in the order R,G,B). No compression is performed on the thumbnail image.

Storing a thumbnail image in the JFIF APP0 marker is now discouraged, though it is still supported for backward compatibility. Version 1.02 of JFIF defines extension markers that allow thumbnail images to be stored separately from the identification marker. This method is more flexible, because multiple thumbnail formats are permitted and because multiple thumbnail images of different sizes could be included in a file. Version 1.02 allows color-mapped thumbnails (one byte per pixel plus a 256-entry colormap) and JPEG-compressed thumbnails, in addition to the 24-bit RGB thumbnail format. In any case, a thumbnail image is limited to less than 64K bytes because it must fit in an APP0 marker.

Following the JFIF marker segment, there may be one or more optional JFIF extension marker segments. Extension segments are used to store additional

Ex.1007 / Page 540 of 1158
APPLE INC.

information and are found only in JFIF version 1.02 and later. The structure of these extension segments is shown below:

```
typedef struct _JFIFExtension
{
BYTE   APP0[2];          /* 00h  Application Use Marker */
BYTE   Length[2];        /* 02h  Length of APP0 Field   */
BYTE   Identifier[5];    /* 04h  "JFXX" (zero terminated) Id String */
BYTE   ExtensionCode;    /* 09h  Extension ID Code      */

} JFIFEXTENSION;
```

APP0 contains the values FFh E0h.

Length stores the length in bytes of the extension segment.

Identifier contains the values 4Ah 46h 58h 58h 00h (JFXX).

ExtensionCode indicates the type of information this extension marker stores. For version 1.02, the only extension codes defined are 10h (thumbnail encoded using JPEG), 11h (thumbnail stored using 1-byte pixels and a palette) and 13h (thumbnail stored using 3-byte RGB pixels).

The extension data follows the extension segment information and varies in size and content depending upon the ExtensionCode value. (Refer to the current JFIF specification for the possible formats of the extension marker segment.)

JFIF decoders must be prepared to ignore unrecognized extension markers and APPn segments. Application-specific APPn markers not recognized by a JPEG decoder can be simply skipped over by using the data length field of the marker.

The JFIF marker is essentially a guarantee that the file conforms to the JFIF conventions. Most JFIF decoders therefore regard the JFIF marker segment as optional, and are quite capable of reading a raw JPEG data stream that complies with the JFIF conventions regarding color space and sample alignment. (There are many such files out there, because JFIF merely formalized common practice in these areas.) A robust decoder will treat a JFIF file as a stream of blocks, with no assumptions about block order beyond those mandated by the JPEG standard. This makes it possible to read many non-standard and incorrect JFIF file variations, such as a COM marker inserted between the SOI and JFIF APP0 markers (there are a fair number of these in existence too). We also recommend that a decoder should accept any JFIF file with a known major version number, even if the minor version number is newer than those known to the decoder.

Ex.1007 / Page 541 of 1158
APPLE INC.

The actual JPEG data in a JFIF file follows all APP0 markers and adheres to the format defined in the JPEG documentation. The baseline JPEG process is the recommended type of image data encoding to be used in JFIF files. This is to ensure maximum compatibility of JFIF files for data interchange.

To identify a JFIF file or data stream, scan for the values FFh D8h FFh. This will identify the SOI marker, followed by another marker. In a proper JFIF file, the next byte will be E0h, indicating a JFIF APP0 marker segment. However, it is possible that one or more other marker segments may be erroneously written between the SOI and JFIF APP0 markers (a violation of the JFIF specification). As previously mentioned, a decoder should still attempt to read the file.

The next two bytes (the APP0 segment length) vary in value, but are typically 00h 10h, and these are followed by the five byte values 4Ah 46h 49h 46h 00h (JFIF). If these values are found, the SOI marker (FFh D8h) marks the beginning of a JFIF data stream. If only the FFh D8h FFh values are found, but not the remaining data, then a "raw" JPEG data stream has been found. All JFIF and JPEG data streams end with the End Of Image (EOI) marker values FFh D9h.

There are many proprietary image file formats which contain JPEG data. Many simply encapsulate a JPEG or JFIF data stream within their own file format wrapper. Scanning for the JPEG SOI marker and reading until the EOI marker is encountered will usually allow you to extract the JPEG/JFIF data stream. At least one proprietary image file format, the .HSI format by Handmade Software, contains JPEG data, but cannot be successfully read or uncompressed without using special software, due to proprietary modifications of the JPEG encoding process. (All .HSI files begin with the values 68h 73h 69h 31h and should not be considered normal JPEG files.)

Only two non-proprietary formats, other than JFIF, currently support JPEG-encoded data. The latest version of the Macintosh PICT format prepends a PICT header to a JFIF file stream. Strip off the PICT header (everything before the SOI marker) and any trailing data (everything after the EOI marker) and you have the equivalent of a JFIF file. The other format, TIFF 6.0, also supports JPEG and is discussed in depth in the article on TIFF.

For Further Information

For further information about the JFIF file format, see the specification included on the CD-ROM that accompanies this book. You may also contact C-Cube Microsystems at:

Ex.1007 / Page 542 of 1158
APPLE INC.

C-Cube Microsystems
Attn: Scott St. Clair
Corporate Communications
1778 McCarthy Blvd.
Milpitas, CA 95035
Voice: 408-944-6300
FAX: 408-944-6314

See also Chapter 9 for information about JPEG compression. The JPEG FAQ and the compression FAQ, also included on the CD-ROM, contain background information about JPEG.

The JPEG standard itself is not available electronically; you must order a paper copy through ISO. In the United States, copies of the standard may be ordered from:

American National Standards Institute, Inc.
Attn: Sales
1430 Broadway
New York, NY 10018
Voice: 212-642-4900

The standard is divided into two parts; Part 1 is the actual specification, and Part 2 covers compliance-testing methods. Part 1 of the draft has now reached International Standard status. See this document:

Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and Guidelines. Document number ISO/IEC IS 10918-1.

Part 2 is still at Committee Draft status. See this document:

Digital Compression and Coding of Continuous-tone Still Images, Part 2: Compliance Testing. Document number ISO/IEC CD 10918-2.

See the discussion of JPEG in Chapter 9 for a list of additional references, including commercially available books that contain the JPEG specification and the JPEG FAQ (Frequently Asked Questions). The CD-ROM contains several JPEG programs and libraries.

Ex.1007 / Page 543 of 1158
APPLE INC.

Kodak Photo CD

NAME:	Kodak Photo CD
ALSO KNOWN AS:	Photo CD
TYPE:	Bitmap
COLORS:	24-bit
COMPRESSION:	Proprietary
MAXIMUM IMAGE SIZE:	2048×3072
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Eastman Kodak
PLATFORM:	All
SUPPORTING APPLICATIONS:	Photo CD Access, Shoebox, Photoshop, others
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Static data storage of multi-resolution deep-pixel images.

COMMENTS: Kodak will not divulge information on the format that would enable developers to directly access the image data.

Overview

Photo CD is actually the name of a CD-ROM-based storage and retrieval system from Eastman Kodak. Most people in the development community use the name Photo CD to refer to the files associated with the system, however, and we will conform to this usage.

A Photo CD CD-ROM is intended for the storage of conventional film-based photographic images which have been converted to digital form, using a slide or flatbed scanner, for instance. This is apparently part of Kodak's strategy for the product: to serve as an adjunct to their conventional film business. There are no technical restrictions on the source of the data, however.

Unfortunately, we cannot describe the Kodak Photo CD format in any detail because Kodak will not divulge the details of the format, and, in fact, has threatened legal action to those who would seek to reverse-engineer the

Ex.1007 / Page 544 of 1158
APPLE INC.

product. This decision on the part of Kodak has enraged members of the development community who have an interest in the future of imaging technology. Cooler heads see Photo CD as a transitional technology. Our own opinion is that a large company with a sufficient presence in the market and a long-term view could have used a system like Photo CD as a way of capturing a major share of the disc-based multimedia market. However, Kodak seems to have taken the more conservative course of protecting their traditional film-based business.

Kodak does, however, sell a shrink-wrapped Photo CD development kit for a reasonable fee, which provides an API of sufficient richness for almost any developer need. At the time of this writing, there are Microsoft Windows, Macintosh, and UNIX versions of the toolkit available. Obviously, developers on other platforms are out of luck, unless Kodak sees fit to accommodate their needs. Toolkits are closely coupled to the platform supported.

As a result of the situation we have described, we are obviously able in this article to provide only information about the Photo CD system and format that is publicly available from Kodak.

File Organization

In the Kodak Photo CD environment, groups of images and associated information written at one time is called a session. The original Photo CD specification called for hardware that supported a single session per disc. Later versions of the Photo CD system allow multiple sessions per disc, which requires special hardware, firmware revisions, or a combination of both to read. As a consequence of this, many older CD-ROM drives will not read multi-session Photo CD discs, so you might make sure that yours does before you get involved with the Photo CD system.

Taking advantage of the storage capacity of CD-ROMs (more than 600 megabytes), images are stored on disc at multiple resolutions, in an arrangement called pyramid encoding. This accomplishes the same thing as the common strategy of storing a "postage stamp," or reduced version of the main image in the same file, albeit carried to a logical extreme. At the time of this writing, six resolutions are normally stored for each image:

Ex.1007 / Page 545 of 1158
APPLE INC.

Base Over 64	64×96
Base Over 16	128×192
Base Over 4	256×384
Base	512×768
Base Times 4	1024×1536
Base Times 16	2048×3072

Another version of the Photo CD product, called Photo CD Pro, may contain higher resolutions, including:

Base Times 16	2048×3072

Multiple versions of the image are grouped into a file that Kodak calls an Image Pac. A copy of at least one of the two lowest-resolution versions of the image in the Image Pac in the current sessions is stored in another file called the Overview Pac. These are used for the display of postage stamp images, which might be used by an application for quick display of the images in the Image Pac, for selection purposes, perhaps.

File Details

In the Kodak Photo CD format, Image data is compressed using a proprietary algorithm. Data is stored in what Kodak calls PhotoYCC format. The developer toolkit delivers color in several formats, depending on the platform. These include 256-level gray-scale and various palette-based formats, in addition to 24-bit YCC and RGB.

In the 24-bit YCC format, 24 bits of data per pixel are distributed among three color components, called Y (luminance information), C1, and C2 (two chrominance channels). Each channel occupies eight bits of data. Although the YCC format has some advantages, most developers choose RGB as the preferred model in which they want the toolkit to deliver the image data.

For Further Information

More details about the Photo CD format are available in descriptive documentation from Kodak marketing sources and in the Kodak Photo CD Access Developer Toolkit for your platform, which contains the following:

Ex.1007 / Page 546 of 1158
APPLE INC.

- A disc full of sample images
- The Access Developer Toolkit Programmer's Guide
- A disk containing the library and associated files needed to compile your application, including a sample application with source code included

For information about obtaining these, contact:

Eastman Kodak Corporation
343 State Street
Rochester, NY 14650
Voice: 800-242-2424
WWW: *http://www.kodak.com/*

As mentioned above, Kodak has threatened legal action against developers who have tried to make details of the Photo CD format public, although the organization has not been completely successful in suppressing information. Source code has been posted to the Internet that will convert Photo CD files to PBM format (used by the pbmplus utilities described in Appendix C, *Installation and Setup*) and presumably remains available at many sites. As a consequence of this posting of information and source code, you may run across an application which reads and manipulates Photo CD format files, but which may not be properly licensed from Kodak. Always check to see if the application vendor is properly licensed.

Ex.1007 / Page 547 of 1158
APPLE INC.

Kodak YCC

NAME:	Kodak YCC
ALSO KNOWN AS:	YCC, ICC
TYPE:	Bitmap
COLORS:	8-bit, 24-bit
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Eastman Kodak
PLATFORM:	All
SUPPORTING APPLICATIONS:	Unknown
SPECIFICATION ON CD:	Yes (summary description by third party)
CODE ON CD:	Yes
IMAGES ON CD:	No
SEE ALSO:	Kodak Photo CD

USAGE: Unknown

COMMENTS: Included because YCC files had some currency in high-end graphics and because of its relevance to the Photo CD format.

Overview

We have been unable to find information on the origin of what has come to be called the Kodak YCC format. Obviously, it originated at Eastman Kodak, in one of the company's graphics-related divisions, but whether it is a "real" format or just a printer dump format, we are unable to tell. We have included it because we have been able to obtain some information on the format and because it may be of interest to people involved with Photo CD or 24-bit color applications in general.

The Kodak YCC format provides data in a format compatible with Kodak's XL7700 printer, which produces truecolor and gray-scale output.

Ex.1007 / Page 548 of 1158
APPLE INC.

File Organization

A Kodak YCC file consists of a header followed by bitmap data. The bitmap data is organized into three planes in the order red, green, and blue.

File Details

The Kodak YCC has the following structure:

```
LONG Magic;          /* Magic number = 5965600 */
LONG HSize;          /* Header Size, in bytes */
CHAR Un01[n];        /* Unused (n = Header Size - 4 bytes) */
LONG HSize;          /* File header length */
LONG FSize;          /* File size */
CHAR FName[16]       /* Filename */
LONG FType;          /* File Type (= 7) */
CHAR. Un02[8];       /* Unused */
LONG XSize;          /* Image X size */
LONG YSize;          /* Image Y size */
CHAR Un03[12];       /* Unused */
LONG Planes;         /* Number of image planes (usually 1 or 3) */
CHAR Un04[8];        /* Unused */
```

Following the header is the image data. If the image is composed of gray-scale data, one plane of 8-bit gray-scale data is present. If the image is truecolor, then there are three planes in the sequence red, green, blue. Each plane consists of 8-bit data. If the data is to be interpreted otherwise, for example, as Y,C,C (luminance followed by two chrominance channels), each channel is eight bits in length, but the rendering application must interpret the data according to the appropriate color model.

For Further Information

For further information about the Kodak YCC format, see the summary description (a newsgroup posting) included on the CD-ROM that accompanies this book. We have tried unsuccessfully to obtain information from Kodak about this format, so we don't believe you will be able to get any help from them. If you wish to try yourself, contact Kodak at:

Eastman Kodak Corporation
343 State Street
Rochester, NY 14650
Voice: 800-242-2424
WWW: *http://www.kodak.com/*

Ex.1007 / Page 549 of 1158
APPLE INC.

Lotus DIF

NAME:	Lotus DIF
ALSO KNOWN AS:	DIF, Data Interchange Format
TYPE:	Vector
COLORS:	NA
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	Unknown
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Software Arts
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Spreadsheets, others
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	Microsoft SYLK

USAGE: Exchange of numerical data often associated with spreadsheets.

COMMENTS: Not usually considered a graphics format but often a carrier of graphics data. Big- and little-endian issues are moot because data is stored as 7-bit ASCII.

Overview

The Lotus DIF (Data Interchange Format) is used for the storage and exchange of numeric data between applications such as spreadsheets. Although DIF is not usually considered a graphics file format, it is vector-based and often carries information used to generate both bitmap and vector images. It is of interest because the data is always stored in ASCII format.

DIF was developed by Software Arts and originally appeared along with the VisiCalc spreadsheet program, which was first released in 1979. Because most spreadsheet applications have their own native file format for storing information, it is usually not possible for a single application to support every other format. DIF has become one of the commonly used interchange formats, perhaps because it has been around so long.

Ex.1007 / Page 550 of 1158
APPLE INC.

A spreadsheet is a 2D matrix of storage cells, each of which contains numeric data, text data, and formulas. Stored along with the data item in each cell is a unique identifier, usually the coordinates associated with the cell itself. Spreadsheet files also may contain information relevant only to the originating application, which is normally ignored by an application seeking to extract the data for other uses.

Popular software applications that support DIF include Lotus 1-2-3 and Borland's Quattro Pro. Since DIF files contain only 7-bit ASCII characters, they can be edited using a simple text editor. DIF is also independent of any hardware issues.

File Organization

A DIF file consists of a header followed by a block of data. The header starts with the ASCII text:

```
TABLE
0,1
"string"
```

where *string* is any ASCII string, often the name of the file or other identifying information. The header ends with the following:

```
DATA
0,0
""
```

Following this is the actual spreadsheet cell data and records containing information to be used in modifying this data.

File Details

This section describes the format of the DIF header and the different types of records.

Header

The header consists of a number of entries, each consisting of a record type keyword, a numeric value, and an optional text string. These are arranged as follows:

```
Record Type
Vector, value
"string"
```


Ex.1007 / Page 551 of 1158
APPLE INC.

The *Record Type* field identifies the data following the field to the rendering or manipulating application. The *Vector* value affects the interpretation of the *value* data and indicates the spreadsheet column to which the *value* data is to apply. A *value* of zero means that the data applies to the entire spreadsheet. The *value* field must be zero if data in the *Vector* field is nonzero. The *string* field is contained in double quotes, with empty double quotes (`""`) indicating an empty string.

Record Types
Possible record types are as follows:

TABLE
VECTORS
TUPLES
DATA
COMMENT
LABEL
UNITS
TRUELENGTH
MINORSTART
MAJORSTART
PERIODICITY
SIZE
DISPLAYUNITS

Only the first four record types must appear in every header: TABLE, VECTORS, TUPLES, and DATA.

Data following TABLE entries has the following format:

```
0,1
"string"
```

where *string* is any ASCII string.

Data following VECTORS entries has the following format:

```
0,columns
""
```

where *columns* is the number of columns in the spreadsheet.

Ex.1007 / Page 552 of 1158
APPLE INC.

Data following TUPLES entries has the following format:

```
0,rows
""
```

where *rows* is the number of rows in the spreadsheet.

Data following DATA entries has the following format:

```
0,0
""
```

and marks the end of the header.

Data following COMMENT entries has the following format:

```
column,lines
"string"
```

which is similar to the definition of LABEL.

Data following LABEL entries has the following format:

```
column,lines
"string"
```

where *column* is the starting column of the LABEL stored as *string*, and *lines* is the number of columns spanned by LABEL, usually one.

Data following UNITS entries has the following format:

```
columns,0
"string"
```

where *string* denotes the units of measurement associated with the values in the *columns* of the spreadsheet.

Data following TRUELENGTH entries has the following format:

```
column-number,rows
""
```

where *rows* is the number of rows containing actual data.

Data following MINORSTART entries has the following format:

```
column-number,time-value
""
```

where *time-value* is the month, day, hour, or second relevant to the start of the data in the column referred to.

Ex.1007 / Page 553 of 1158
APPLE INC.

Data following MAJORSTART entries has the following format:

```
column-number,first-year
""
```

where *first-year* denotes the year data in the column referred to.

Data following PERIODICITY entries has the following format:

```
columns,period
""
```

where *period* is the time duration of time-oriented data.

Data following SIZE entries has the following format:

```
column-number,bytes
""
```

where *bytes* is the width of *columns* in bytes.

Data following DISPLAYUNITS entries has the following format:

```
columns,0
"string"
```

where *string* denotes information associated with display devices.

Data

Data record structures have the following format:

```
data-type,data
"string"
```

data-type can be SPECIAL, NUMERIC, and STRING, denoted by –1, 0, and 1, respectively.

SPECIAL data appears as follows:

```
-1,0
BOT
 ⋮
-1,0
EOD
```

where BOT and EOD are strings (without quotes) denoting beginning-of-table and end-of-data, respectively.

Ex.1007 / Page 554 of 1158
APPLE INC.

NUMERIC data appears as follows:

```
0,data
value-indicator
```

where *value-indicator* indicates the type of data stored in *data*:

TRUE	1
FALSE	0
V	any numeric value
NA	not known
ERROR	0

String data appears as follows:

```
1,0
"string"
```

where *string* is any text string.

For Further Information

For further information about the Lotus DIF file format, you might try contacting Lotus at:

Lotus Development Corporation
55 Cambridge Parkway
Cambridge, MA 02142
Voice: 617-577-8500
Voice: 800-831-9679
FAX: 617-225-1197

Information for this article came from the DIF Technical Specification published by:

DIF Clearinghouse
P.O. Box 638
Newton Lower Falls, MA 02162

The Clearinghouse apparently no longer exists, but you may be able to find this document in archives and libraries under their authorship. It is our understanding that responsibility for DIF maintenance has devolved to Lotus, though it is not clear whether Lotus owns or has access to the original DIF Clearinghouse documents. Lotus, of course, is now owned by IBM.

Ex.1007 / Page 555 of 1158
APPLE INC.

An interesting account of the DIF format, by Candace Kalish and Malinda Mayer, was published in the November 1981 issue of *byte* magazine.

The following publications also contain information about DIF:

Beil, Donald H., *The DIF File*, Reston Publishing Co., Reston, VA, 1983.

Walden, Jeffrey B., *File Formats for Popular PC Software*, John Wiley & Sons, New York, NY, 1986.

Ex.1007 / Page 556 of 1158
APPLE INC.

NAME:	Lotus PIC
ALSO KNOWN AS:	Lotus Picture, PIC
TYPE:	Vector
COLORS:	6
COMPRESSION:	NA
MAXIMUM IMAGE SIZE:	Apparently 64K×64K
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Lotus Development
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Lotus 1-2-3 and competing programs, word-processing and desktop-publishing applications, others
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	Microsoft SYLK

USAGE: Used by the graphing program associated with Lotus 1-2-3

COMMENTS: A widely used format for interchange of data, primarily business graphics. Somewhat dated. Big-endian in format, although originating under MS-DOS on Intel-based machines.

Overview

Lotus PIC appeared in support of early versions of Lotus 1-2-3; files in PIC format were generated by the main application for use by an auxiliary program called Lotus Print Graph. Although recent versions of the application still support PIC, they have started using the Computer Graphics Metafile (CGM) as well, and we can safely assume that the days of PIC are numbered. Nevertheless, a great deal of data still exists in PIC format.

Ex.1007 / Page 557 of 1158
APPLE INC.

File Organization and Details

The file is very simple and consists of a header, vector data, and an end-of-file indicator. The header appears to be arbitrary and contains the following hex string:

```
01 00 00 00 01 00 08 00 44 00 00 00 0C 7F 09 06
```

Following the header is a list of encoded drawing commands, stored either as byte pairs or as 16-bit values. Either form may be followed by arguments. Commands are recognized by reading data either one byte at a time, assembling 16 bits of data in memory, or reading data 16 bits at a time and examining the first byte of each item. Coordinate values are always stored as 16-bit signed integers. Although positional data can theoretically be in the range -32,767 to 32,767, Lotus Print Graph always scales data to fit into the rectangle 0, 0, 3200, 2311.

Drawing commands supported by PIC are listed below:

BN	color	N is an 8-bit color value
A0 *XX YY*	move	Move drawing cursor to XX,YY
A2 *XX YY*	draw	Draw to XX,YY, update cursor
30 *N-1 X1 Y1 . . . XN YN*	fill	Filled polygon of N vertices
D0 *N-1 X1 Y1 . . . XN YN*	fill outlined	Filled polygon with outline
AC *XX YY*	text size	XX and YY are char cell size
A7 *N*	font	Set font: type 0 or 1 only
A8 *N STRING*	text	Draw NULL-terminated text string STRING, N contains direction and alignment information: 00 horizontal 10 vertical up 20 upside down 30 vertical down 00 center aligned 01 left center aligned 02 top center aligned 03 right center aligned 04 bottom center aligned 05 top left aligned

Ex.1007 / Page 558 of 1158
APPLE INC.

		06 top right aligned
		07 bottom left aligned
		08 bottom right aligned
60-6F	end	End of image

The following example draws a line from 0,0 to 100,100 and draws the string "text" with characters fitting into an 8 by 10 cell:

```
A0 00 00 00 00 A2 00 64 00 64 AC 00 08 00 0A A8 00 74 65 78 74 60
```

For Further Information

Lotus no longer supports PIC, so it is difficult to get information about it. You might try contacting Lotus at:

Lotus Development Corporation
55 Cambridge Parkway
Cambridge, MA 02142
Voice: 617-577-8500
Voice: 800-831-9679
FAX: 617-225-1197

The following book, available in bookstores or from Lotus, provides additional information about Lotus PIC:

Lotus Development Corporation, *Lotus File Formats for 1-2-3, Symphony, & Jazz*, Lotus Books, Cambridge, MA, 1986.

Ex.1007 / Page 559 of 1158
APPLE INC.

Lumena Paint

NAME:	Lumena Paint
ALSO KNOWN AS:	.PIX, .BPX
TYPE:	Bitmap
COLORS:	24-bit maximum
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	Unlimited
MAXIMUM IMAGE SIZE:	Unlimited
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Time Arts Inc.
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Lumena, others
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	TGA

USAGE: Used mainly in conjunction with Time Arts programs, particularly Lumena Paint.

COMMENTS: This format is used frequently by production houses for data interchange.

Overview

Lumena Paint is a 24-bit paint program for the PC that uses the TGA, TIFF, and EPS file formats to import and export images. Lumena also uses its own native bitmap formats, which often have the extension .PIX (Time Arts Picture Format) and .BPX (Time Arts Big Picture Format), and which are also referred to as the Lumena 16 and Lumena 32 formats, respectively.

Lumena sold into what was known as the Targa market, centered around compatibility with truecolor display adapters for the PC sold by Truevision and its competitors. This was a small niche market in the PC world, and at its peak consisted of about 50,000 sites. Increased interest in truecolor on PCs led Hercules to introduce a relatively inexpensive display adapter (Hercules Graphics Station) supporting 24-bit color, and as part of its marketing effort, enter into an OEM arrangement with Time Arts to bundle Lumena with the boards.

Ex.1007 / Page 560 of 1158
APPLE INC.

File Organization

Besides the depth of the pixel data, the main difference between the Lumena 16 and the Lumena 32 formats is in the header. Both formats have the same file header. In the Lumena 16 format, four of the header fields associated with pixel values are two bytes in size, while the same fields are four bytes in size for the Lumena 32 format. Lumena 16 files may also contain a postage stamp image, while Lumena 32 files may not.

Following the common file header may be a descriptor header; this header is different for the two formats. Lumena 16 files have a PIX header, whereas Lumena 32 files have a BPX header.

File Details

Each pixel value in the Lumena 16 image is two bytes in size (five bits each of red, green, and blue, and one overlay bit), and each pixel value in a Lumena 32 file is 32 bits (eight bits each of red, green, blue, and alpha channel). Both the alpha channel and the overlay bits occupy the most significant bits of each pixel value. Image data always follows the postage stamp data, or a feature called the descriptor header, if no postage stamp image is present in the file.

The largest size of a typical Lumena 16 image is 512×482 pixels. Larger images are stored using the Lumena 32 format and at a size that is an exact multiple of the display resolution (for example 1024×768×32 bits).

Both Lumena file types start with the same file header:

```
typedef struct _LumenaHeader
{
  BYTE  DescriptorSize;     /* Size of Image Descriptor   */
  BYTE  IsImageStamp;       /* Image Stamp Present   */
  BYTE  FileType;           /* File Encoding Type   */
  WORD  StampWidth;         /* Stamp Width in Pixels   */
  WORD  StampHeight;        /* Stamp Length in Pixels   */
  BYTE  StampBPP;           /* Bits per Pixel in Stamp   */
  WORD  XOrigin;            /* X Origin of Image   */
  WORD  YOrigin;            /* Y Origin of Image   */
  WORD  ImageWidth;         /* Image Width in Pixels   */
  WORD  ImageHeight;        /* Image Height in Pixels   */
  BYTE  BitsPerPixel;       /* Number of Bits Per Pixel   */
  BYTE  AlphaMaskBPP;       /* Alpha Bits Per Pixel   */

} LUMENAHEAD;
```


Ex.1007 / Page 561 of 1158
APPLE INC.

DescriptorSize is the size of the descriptor header in bytes. The descriptor header is a second header that may follow the file header and that differs in size between the two Lumena formats.

IsImageStamp field is set to 1h if there is a postage-stamp image included in the file, otherwise it is set to 0h. The postage-stamp image always follows the descriptor header.

A postage-stamp image is a smaller version of the primary image stored in the Lumena file. Postage stamps are used to preview the contents of an image file without taking the time to display the original image. Postage-stamp images may be very quickly displayed because of their small size. Typically, postage stamps are one-eight the height and width of the original image, with 64×64 pixels being a typical maximum size. A typical 512×482-pixel Lumena image would then contain a 64×60-pixel postage-stamp image. Although postage stamps are useful, not every Lumena image file will contain one. (See the article on the TGA format for information on ways to create postage-stamp images.)

FileType indicates the type of data compression algorithm used on the image data. A value of 02h indicates a standard file with uncompressed image data; the image data is arranged just as it is in the TGA Type 2 data format. A value of 0Ah indicates run-length encoded image data the same as found in TGA Type 10 image data. A value of 8Eh indicates an older style of data compression that is no longer used by applications supporting the Lumena image file formats.

StampWidth, StampHeight, and StampBPP all store information about the postage stamp image. Only Lumena 16 image files may contain postage-stamp images; Lumena 32 image files never contain these images. The StampWidth and StampHeight are in pixels, and the StampBPP field contains the number of bits per pixel in the postage-stamp data, including alpha channel bits, if any.

XOrigin and YOrigin are the starting coordinates of the image on the display with 0,0 being the lower-left corner of the screen.

ImageHeight and ImageWidth represent the size of the image in pixels.

BitsPerPixel contains the number of bits in each pixel of image data. This value is 16 for Lumena 16 images and 32 for Lumena 32 images and includes any alpha bits present in the pixel data.

AlphaMaskBPP contains the number of bits in each pixel used for alpha channel data. This value is 0 for Lumena 16 images and 8 for Lumena 32 images.

Ex.1007 / Page 562 of 1158
APPLE INC.

Lumena 16 files also have a PIX descriptor header in the following format:

```
typedef struct _Lumena16Descriptor
{
WORD  Identifier;     /* Descriptor Identifier   */
WORD  RedMask;        /* Mask for Red Bits   */
WORD  GreenMask;      /* Mask for Green Bits   */
WORD  BlueMask;       /* Mask for Blue Bits   */
WORD  XAspectRatio;   /* X Axis Image Aspect Ratio   */
WORD  YAspectRatio;   /* Y Axis Image Aspect Ratio   */
WORD  Background;     /* Background Color (Black = 0)  */
BYTE  Comment[40];    /* Text Comment   */

} LUM16DESCRP;
```

Lumena 32 files have a BPX descriptor header instead of a PIX header. The BPX descriptor header is identical to the PIX descriptor header except for four fields that are four bytes in size rather than two:

```
typedef struct _Lumena32Descriptor
{
WORD   Identifier;    /* Descriptor Identifier   */
DWORD  RedMask;       /* Mask for Red Bits   */
DWORD  GreenMask;     /* Mask for Green Bits   */
DWORD  BlueMask;      /* Mask for Blue Bits   */
WORD   XAspectRatio;  /* X Axis Image Aspect Ratio   */
WORD   YAspectRatio;  /* Y Axis Image Aspect Ratio   */
DWORD  Background;    /* Background Color (Black = 0)     */
BYTE   Comment[40];   /* Text Comment   */

} LUM32DESCRP;
```

Identifier always contains the value 8Eh and is used to identify the start of the descriptor header.

RedMask, GreenMask, and BlueMask contain the values used to mask and shift out the separate red, green, and blue values from the pixel data. The code to do so is shown below:

```
RedValue   = (PixelValue & RedMaskValue) >> (BitsPerPixel * 2);
GreenValue = (PixelValue & GreenMaskValue) >> (BitsPerPixel * 1);
BlueValue  = (PixelValue & BlueMaskValue) >> (BitsPerPixel * 0);
```

XAspectRatio and YAspectRatio contain the horizontal and vertical aspect ratios of the image.

Background indicates the background color of the display not covered by the image. The default is 0 for black, and this value may be any valid screen color value.

Ex.1007 / Page 563 of 1158
APPLE INC.

Comment is a NULL-terminated string of ASCII characters. Images created using the Lumena Paint program typically contain the comment "Time Arts Lumena file."

For Further Information

For further information about the Lumena Paint format and application, see the specification included on the CD-ROM that accompanies this book. You may also contact:

Time Arts Inc.
Attn: Scott Gross
Vice-President, Engineering
1425 Corporate Center Parkway
Santa Rosa, CA 95407
Voice: 800-959-0509
FAX: 707-576-7731
BBS: 707-576-7352

You might also be able to obtain information about Lumena from Hercules Computer, which bundles Lumena Paint with its PCs and graphics boards. Contact:

Hercules Computer Inc.
Attn: Lumena/Hercules Art Department
921 Parker Street
Berkeley, CA 94710
Voice: 510-540-6000
Voice: 800-532-0600
FAX: 510-540-6621
BBS: 510-540-0621

Ex.1007 / Page 564 of 1158
APPLE INC.

NAME:	Macintosh Paint
ALSO KNOWN AS:	PNTG, MAC, MacPaint
TYPE:	Bitmap
COLORS:	Mono
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	576×720 pixels
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Apple Computer Inc.
PLATFORM:	Macintosh
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	Yes (for Atari)
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	Macintosh PICT

USAGE: Clip art, screen dumps, monochrome artwork

COMMENTS: A well-defined, well-documented format in use on the Macintosh platform. MacPaint is quick and easy to read and decompress, but it lacks support for more than monochrome images. Numbers in the file are stored in big-endian format.

Overview

Macintosh Paint (MacPaint) is the original and most common graphics file format used on the Apple Macintosh. Most Macintosh applications that use graphics are able to read and write the MacPaint format. MacPaint files on the Macintosh have the file type PNTG, while on the PC they usually have the extension .MAC. The first real image files widely available to PC users were MacPaint files. PC users usually obtained them from BBSs or shareware disks, and a number of programs exist that allow MacPaint files to be displayed and printed using a PC under MS-DOS. Today, extensive black-and-white clip art and graphics are available in the MacPaint format. MacPaint files are also used to store line drawings, text, and scanned images.

Ex.1007 / Page 565 of 1158
APPLE INC.

MacPaint images are always black and white and are a fixed size (576 pixels wide by 720 scan lines high) and fixed resolution (75 dpi). Uncompressed, the image data is always 51,840 bytes in size. Because the Apple Macintosh is based on the Motorola 68000 series of CPUs, most files are stored in big-endian format, and MacPaint files are no exception. They are always read and written in big-endian format, no matter what the host platform. The ENDIANIO library can be used to read and write MacPaint files on non-big-endian systems.

Files are stored differently on the Macintosh than they are on most other systems in common use. Every Macintosh file consists of two parts, called forks. Although a user sees only a single file, data is actually stored as two physical files on disk. The first file is called the data fork, which stores program information. The second file is called the resource fork, and it stores program code. Data associated with a MacPaint file occupies only the data fork of the file pair; its companion resource fork is always empty.

Outside the Macintosh environment a MacPaint file is stored as a single file, with the two forks combined into one file, allowing it to reside on foreign file systems not adhering to the Macintosh conventions. A MacBinary header is prepended to the file. The MacBinary header is a structure which allows a Macintosh file to be copied or otherwise transported between a Macintosh and another system, and which contains the information required to reconstruct the two forks when the file is returned to the Macintosh environment. It is necessary to preserve the MacBinary header only if the file will one day be returned to a Macintosh environment; otherwise, it can be stripped from the file.

There are actually two MacBinary standards, the original MacBinary and MacBinary II. Both standards have a header that is 128 bytes in length. The MacBinary II header contains additional information not found in the original MacBinary header.

File Organization

The structure of a MacBinary II header is shown below:

```
typedef struct _MacBinaryIIHeader
{
BYTE   Version;              /* Always set to 0 */
BYTE   FileNameLength;       /* Size of file name (0 to 31) */
BYTE   FileName[63];         /* File name */
DWORD  FileType;             /* Type of Macintosh file */
DWORD  FileCreator;          /* ID of program that created file */
BYTE   FileFlags;            /* File attribute flags */
```


Ex.1007 / Page 566 of 1158
APPLE INC.

```
BYTE    Reserved1;              /* Reserved field */
WORD    FileVertPos;            /* File vertical position in window */
WORD    FileHorzPos;            /* File horizontal position in window */
WORD    WindowId;               /* Window or folder ID */
BYTE    Protected;              /* File protection (1 = protected) */
BYTE    Reserved2;              /* Reserved field */
DWORD   SizeOfDataFork;         /* Size of file data fork in bytes */
DWORD   SizeOfResourceFork;     /* Size of file resource fork in bytes */
DWORD   CreationStamp;          /* Time and date file created */
DWORD   ModificationStamp;      /* Time and date file last modified */
WORD    GetInfoLength;          /* GetInfo message length */

/* The following fields were added for MacBinary II */

WORD    FinderFlags;            /* Finder flags */
BYTE    Reserved3[14];          /* Reserved field */
DWORD   UnpackedLength;         /* Total unpacked file length */
WORD    SecondHeadLength;       /* Length of secondary header */
BYTE    UploadVersion;          /* MacBinary version used with uploader */
BYTE    ReadVersion;            /* MacBinary version needed to read */
WORD    CrcValue;               /* CRC value of previous 124 bytes */
BYTE    Reserved4[2];           /* Reserved field */

} MACBIN2HEAD;
```

Before extracting the image data from a MacPaint file in a non-Macintosh environment, you must determine if a MacBinary header is prepended. This is best done by reading the bytes at offsets 101 through 125 and checking to see if they are all zero. The byte at offset 2 should be in the range of 1 to 63, and the DWORDs at offsets 83 and 87 should be in the range of 0 to 007FFFFFh. If all of these checks are true, then a MacBinary header is present.

It is not necessary for a non-Macintosh application to modify the MacBinary header unless the image data is changed or the MacPaint file has been created outside of the Macintosh environment with the intent of one day being returned to the Mac. However, it is good general practice to assume that your image file will one day return to the originator platform. Because any application reading a MacPaint file must be prepared to decode the MacBinary header anyway, there is no good reason for omitting it or for failing to update its fields when the file is changed.

File Details

Version, the first byte of a MacBinary header, is always zero; MacPaint files with or without a MacBinary header, always start with a zero byte. In fact, if the first byte is not zero, do not treat the file as a MacPaint file.

Ex.1007 / Page 567 of 1158
APPLE INC.

FileNameLength stores the length of the Macintosh-format filename, which can be from 1 to 63.

FileName stores the actual filename, and only the first "FileNameLength" characters are significant. Note that the filename is not NULL-terminated. Because the Macintosh can accommodate longer filenames than are found on some systems, a certain amount of intelligence is needed when you copy MacPaint files to filesystems that cannot accommodate the full filename. UNIX and Macintosh programmers, in particular, should be wary of copying files to MS-DOS systems and are advised to keep the filenames limited to eight characters or less. Files destined only for UNIX systems should limit names to 14 or fewer characters.

FileType contains up to four ASCII characters indicating the type of file that is attached to the header. A MacPaint file has a type of PNTG; a PICT file (another Macintosh file type) has type PICT; a TIFF file has type TIFF; and so on.

FileCreator also contains a 4-character ASCII identifier that identifies the creator application. The creator identifier is MPNT for MacPaint files created by the MacPaint paint program, for instance.

FileFlags contains file attributes specific to the Macintosh environment; these are represented by the following bits in the field:

Bit 0 Inited
Bit 1 Changed
Bit 2 Busy
Bit 3 Bozo
Bit 4 System
Bit 5 Bundle
Bit 6 Invisible
Bit 7 Locked

FileVertPos and FileHorzPos contain the position of the file on the display screen.

The WindowID and Protected bit flags are specific to the Macintosh environment.

SizeOfDataFork is the size of the MacPaint file minus the size of the MacBinary header.

Ex.1007 / Page 568 of 1158
APPLE INC.

SizeOfResourceFork is always zero for MacPaint files.

CreationStamp and ModificationStamp contain the time and date the MacPaint file was first created and last modified, respectively. The stamp values are stored as the number of seconds since January 1, 1904.

GetInfoLength contains the length of the Get Info comment and is set to zero in MacPaint files.

The following fields were added by the MacBinary II standard:

FinderFlags contains the first eight bit flags of the Finder. Finder bit flags 8 through 15 are stored in the FileFlags field.

UnpackedLength is the uncompressed size of the file.

SecondHeadLength holds the length of any additional header following the MacBinary header; this value is for future expansion of the MacBinary header and is currently set to zero.

UploadVersion and ReadVersion contain version numbers of the programs required to transmit and read the MacBinary II header.

CrcValue contains a value that may be used to check the validity of the first 124 bytes of the header and needs to be recalculated if the header is changed. If this field is set to zero, ignore it.

There are four fields in the MacBinary II header marked as reserved. They are used for padding and as space for additional fields in future revisions of the MacBinary header. They should be set to zero, as should all unused fields in the header.

The MacBinary header is followed by four bytes of data (00h, 00h, 00h, 02h) signaling the start of the actual MacPaint file. Following these four bytes are 304 bytes of pattern data. This data is used and modified by paint programs such as MacPaint as pattern palette data and is not used for the reconstruction or display of MacPaint images themselves. There is always data for 38 patterns, and each pattern is eight bytes in length.

Following the pattern data are 204 bytes of zero-byte data used for padding. The MacPaint image data follows this padding and always starts at file offset 640 when a MacBinary header is present. Image data in a MacPaint file is always compressed using a simple byte-wise run-length encoding (RLE) scheme. Each scan line is always 72 bytes in length and there are always 720 scan lines per MacPaint image.

Ex.1007 / Page 569 of 1158
APPLE INC.

A byte is read and used as the run count. If the most significant bit is set to 1, the byte is converted to its two's-complement value, and the next byte is repeated RunCount times. If the most significant bit is zero, then one is added to the count and the next RunCount bytes are read. We can use the steps shown in the following pseudocode to decode a scan line:

```
Read a byte value
  If high bit is one
     Count is two's complement of byte (count = ~byte value)
     Read a byte
     Write this byte 'count' times
If high bit is zero
     Count is byte value plus one (count = byte value + 1)
     Read and copy the next 'count' bytes
If 72 bytes have been written, the scan line is done
```

Note that the Macintosh displays black characters on a white background, as opposed to the PC and other systems, which display white characters on a black background; for this reason, it may be necessary to flip the bit values of the image data to obtain the proper color orientation.

For Further Information

For further information about the Macintosh Paint format, see the code examples included on the CD-ROM that accompanies this book. You can also contact:

Apple Computer Inc.
20525 Mariani Avenue
Cupertino, CA 95104
Voice: 408-996-1010
Voice: 800-538-9696
FAX: 408-974-1725
WWW: *http://www.apple.com/*

Additional information on this format can be found in:

Apple Computer, *Inside Macintosh: Imaging*, Volumes I and V, Addison-Wesley, Reading, MA, 1985.

These volumes are also available on the Apple Developer CDs.

Ex.1007 / Page 570 of 1158
APPLE INC.

Additional references include:

"MacPaint Documents Format," *Macintosh Technical Note #86,* Apple Computer Developer Technical Support.

Birse, Cameron, Guillermo Ortiz, and Jon Zap. "Things You Wanted to Know About PackBits," *Macintosh Technical Note #71,* Apple Computer Developer Technical Support.

Ex.1007 / Page 571 of 1158
APPLE INC.

Macintosh PICT

NAME:	Macintosh PICT
ALSO KNOWN AS:	PICT, Macintosh Picture, .PCT, QuickDraw Picture Format
TYPE:	Metafile
COLORS:	Up to 24-bit
COMPRESSION:	PackBits, JPEG
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Big-endian
ORIGINATOR:	Apple Computer, Inc.
PLATFORM:	Apple Macintosh
SUPPORTING APPLICATIONS:	Most Macintosh programs
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes
IMAGES ON CD:	Yes
SEE ALSO:	Macintosh Paint

USAGE: Desktop publishing, paint, and imaging applications using QuickDraw calls.

COMMENTS: A versatile format in wide use on the Macintosh by applications having anything to do with graphics. Because of its complexity, however, it is seldom supported on other platforms.

Overview

The Macintosh PICT (Macintosh Picture) format is associated with applications on the Macintosh and is one of the best supported formats on that platform. PICT files are meant to encapsulate the functionality of QuickDraw, the native graphics drawing protocol on the Macintosh, and consist mainly of QuickDraw calls arranged in no particular order. There have been two major releases of QuickDraw, v1.0 and v2.0 (Color QuickDraw). There have also been numerous minor QuickDraw revisions, each associated with a corresponding Macintosh PICT version.

QuickDraw v1.0 supports monochrome bitmaps up to 32K in size. Image resolution is fixed at the original Macintosh display resolution, or 72 dpi.

Ex.1007 / Page 572 of 1158
APPLE INC.

QuickDraw v2.0, sometimes known as Color QuickDraw, supports 8-bit bitmaps as well as monochrome. There is no compression available for 8-bit Version 2.0 PICT files.

All information in Macintosh PICT files is stored in the data fork of the Macintosh file pair. Although the resource fork may be present, it is left empty. Image data is stored in binary format and consists of a series of operators and associated data.

High-level routines in the Macintosh ToolKit are available to read and write PICT files and are often used when writing applications that translate PICT files to other image file formats.

File Organization

All Macintosh PICT files start with a 512-byte header, which contains information that the Macintosh uses to keep track of the file. This is followed by three fields describing the image size (picSize), the image frame (picFrame), and a version number. In v2.0 files, another header follows. In both versions, the preceding information is followed by the image data. In all versions, the end of the file is signalled by an end-of-file operator.

File Details

QuickDraw, and consequently the Macintosh PICT format, is far too complex for us to do justice to it here, so we will merely note some details about the start of the file. A good deal of information and codes are included on the CD-ROM. Note that most secondary references only give examples of bitmap encoding and ignore the vector nature of the format.

The information following the platform-specific 512-byte header is in the following format:

```
SHORT    File size in bytes
SHORT    Frame y-value of top left of image (at 72 dpi)
SHORT    Frame x-value of top left of image (at 72 dpi)
SHORT    Frame y-value of lower right of image (at 72 dpi)
SHORT    Frame x-value of lower right of image (at 72 dpi)
```

in v1.0 files, this is followed by:

```
BYTE     Version operator(0x11)
BYTE     Version number(0x01)
```


Ex.1007 / Page 573 of 1158
APPLE INC.

or, in v2.0 files, by:

```
SHORT    Version operator (0x0011)
SHORT    Version number (0x02ff)
```

Version 2.0 files also have a 26-byte header following the version information:

```
SHORT    Header opcode for Version 2 (0C00)
SHORT    FFEF or FFEE
SHORT    Reserved (0000)
LONG     Original horizontal resolution in pixels/inch
LONG     Original vertical resolution in pixels/inch
SHORT    Frame upper left y original resolution
SHORT    Frame upper left x at original resolution
SHORT    Frame lower right y at original resolution
SHORT    Frame lower right x at original resolution
LONG     Reserved
```

picSize and picFrame records follow the header.

picSize

```
WORD     Picture size in bytes
WORD     Image top
WORD     Image left
WORD     Image bottom
WORD     Image right
```

picFrame (PICT v1.0)

```
BYTE     Version (11h)
BYTE     Picture version (01h)
```

This is followed by the image data. Each record in a PICT version 1 file consists of a one-byte opcode followed by the actual data.

picFrame (PICT v2.0)

```
WORD     Version (0011h)
WORD     Picture version (02ffh)
WORD     Reserved header opcode (0c00h)
WORD     Header opcode (0c00h)
DWORD    Picture size (bytes)
DWORD    Original horizontal resolution (pixels/inch)
DWORD    Original vertical resolution (pixels/inch)
WORD     y value of top left of image
WORD     x value of top left of image
WORD     y value of lower right of image
WORD     x value of lower right of image
DWORD    Reserved
```

This is followed by the image data. Each record of a PICT v2.0 file consists of a two-byte opcode followed by the actual data. Note that opcodes and data must

Ex.1007 / Page 574 of 1158
APPLE INC.

be aligned on 16-byte boundaries, and that certain opcodes in PICT v1.0 and v2.0 files are interpreted differently.

For Further Information

For further information about the Macintosh PICT format, see the documentation and sample code included on the CD-ROM that accompanies this book.

Additional information on the Macintosh PICT format may be obtained from Claris Corporation, a software spinoff from Apple, in the form of an update to Apple Technical Note #27. Apple Technical Notes may be obtained from Apple Computer and from many online information services. Contact:

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95104
Voice: 408-996-1010
Voice: 800-538-9696
FAX: 408-974-1725
WWW: *http://www.apple.com/*

Claris Corporation
5201 Patrick Henry Drive
P.O. Box 58168
Santa Clara, CA 95052-8168
Technical Support: 408-727-9054
Customer Relations: 408-727-8227
WWW: *http://www.claris.com/*

Other Apple Technical Notes related to Macintosh PICT and other Apple formats include:

TN #021 *QuickDraw Picture Definitions*

TN #041 *Offscreen Bitmaps*

TN #091 *Optimizing of the LaserWriter—Picture Comments*

TN #119 *Color QuickDraw*

TN #120 *Offscreen PixMap*

TN #171 *Things You Wanted to Know About PackBits*

TN #181 *Every Picture (Comment) Tells Its Story, Don't It?*

Ex.1007 / Page 575 of 1158
APPLE INC.

TN #154 *Displaying Large PICT Files*

TN #275 *32-Bit QuickDraw Version 1.2 Features*

Additional information on the PICT format can be found in:

Apple Computer, *Inside Macintosh*, vols. I, V, and VI, Addison-Wesley, Reading, MA, 1985.

These volumes are also available on the Apple Developer CDs.

Ex.1007 / Page 576 of 1158
APPLE INC.

NAME:	Microsoft Paint
ALSO KNOWN AS:	MSP
TYPE:	Bitmap
COLORS:	Mono
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	64K×64K pixels
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Microsoft Corporation
PLATFORM:	Microsoft Windows, MS-DOS
SUPPORTING APPLICATIONS:	Microsoft Paint, others
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	Microsoft Windows Bitmap

USAGE: Black-and-white drawings, clip art

COMMENTS: A format that was in wider use in the early days of Microsoft Windows. It is a simple format that is not currently suitable for deep pixel or truecolor images.

Overview

The Microsoft Paint (MSP) image file format is used exclusively for storing black-and-white images. The vast majority of MSP files contain line drawings and clip art. MSP is used most often by Microsoft Windows applications, but may be used by MS-DOS-based programs as well. The Microsoft Paint format is apparently being replaced by the more versatile Microsoft Windows BMP format; it contains information specifically for use in the Microsoft Windows operating environment. For information on the Windows-specific use of the header information, refer to the Microsoft Paint format specification available from Microsoft.

Ex.1007 / Page 577 of 1158
APPLE INC.

File Organization

The Microsoft Paint header is 32 bytes in length and has the following structure. In the discussion that follows, a WORD is a 16-bit unsigned value.

```
typedef struct _MicrosoftPaint
{
WORD  Key1;             /* Magic number     */
WORD  Key2;             /* Magic number     */
WORD  Width;            /* Width of the bitmap in pixels   */
WORD  Height;           /* Height of the bitmap in pixels   */
WORD  XARBitmap;        /* X Aspect ratio of the bitmap   */
WORD  YARBitmap;        /* Y Aspect ratio of the bitmap   */
WORD  XARPrinter;       /* X Aspect ratio of the printer   */
WORD  YARPrinter;       /* Y Aspect ratio of the printer   */
WORD  PrinterWidth;     /* Width of the printer in pixels   */
WORD  PrinterHeight;    /* Height of the printer in pixels   */
WORD  XAspectCorr;      /* X aspect correction (unused)     */
WORD  YAspectCorr;      /* Y aspect correction (unused)     */
WORD  Checksum;         /* Checksum of previous 24 bytes   */
WORD  Padding[3];       /* Unused padding    */

}MSPHEAD;
```

File Details

In the Microsoft Paint header, Key1 and Key2 contain identification values used to determine the version of the file format. For version 1.x of the Microsoft Paint format, the values of the Key1 and Key2 fields are 6144h and 4D6Eh respectively. For version 2.0, the Key1 and Key2 field values are 694Ch and 536Eh respectively.

Width and Height are the size of the bitmap in pixels. The size of the bitmap in bytes is calculated by dividing Width by 8 and multiplying it by Height.

XARBitmap and YARBitmap contain the aspect ratio in pixels of the screen used to create the bitmapped image.

XARPrinter and YARPrinter contain the aspect ratio in pixels of the output device used to render the bitmapped image. When an MSP file is created by a non-Windows application, these four fields typically contains the same values as the Width and Height fields.

PrinterWidth and PrinterHeight contain the size in pixels of the output device for which the image is specifically formatted. Typical values for these fields are the same values as those stored in Width and Height.

Ex.1007 / Page 578 of 1158
APPLE INC.

XAspectCorr and YAspectCorr are used to store aspect ratio correction information, but are not used in version 2.0 or earlier versions of the Microsoft Paint format and should be set to 0.

Checksum contains the XORed values of the first 12 WORDs of the header. When an MSP file is read, the first 13 WORDs, including the Checksum field, are XORed together, and if the resulting value is 0, the header information is considered valid.

Padding extends the header out to a full 32 bytes in length and is reserved for future use.

The image data directly follows the header. The format of this image data depends upon the version of the Microsoft Paint file. For image files prior to version 2.0, the image data immediately follows the header. There are eight pixels stored per byte, and the data is not encoded.

Each scan line in a version 2.0 or later Microsoft Paint bitmap is always RLE-encoded to reduce the size of the data. Each encoded scan line varies in size depending upon the bit patterns it contains. To aid in the decoding process, a scan-line map immediately follows the header. The scan-line map is used to seek to a specific scan line in the encoded image data without needing to decode all image data prior to it. There is one element in the map per scan line in the image. Each element in the scan-line map is 16 bits in size and contains the number of bytes used to encode the scan line it represents. The scan-line map starts at offset 32 in the MSP file and is sizeof(WORD).

Consider the following example. If an application needs to seek directly to the start of scan-line 20, it adds together the first 20 values in the scan-line map. This sum is the offset from the beginning of the image data of the 20th encoded scan line. The scan-line map values can also be used to double-check that the decoding process read the proper number of bytes for each scan line.

Following the scan-line map is the run-length encoded monochrome bitmapped data. A byte-wise run-length encoding scheme is used to compress the monochrome bitmapped data contained in an MSP-format image file. Each scan line is encoded as a series of packets containing runs of identical byte values. If there are very few runs of identical byte values, or if all the runs are very small, then a way to encode a literal run of different byte values may be used.

Ex.1007 / Page 579 of 1158
APPLE INC.

The following pseudocode illustrates the decoding process:

```
Read a BYTE value as the RunType
  If the RunType value is zero
    Read next byte as the RunCount
    Read the next byte as the RunValue
    Write the RunValue byte RunCount times
  If the RunType value is non-zero
    Use this value as the RunCount
    Read and write the next RunCount bytes literally
```

As you can see, this is yet another variation of a simple run-length encoding scheme. A byte is read, and if it contains a value of 0, then the following byte is the RunCount (the number of bytes in the run). The byte following the RunCount is the RunValue (the value of the bytes in the run). If the byte read is non-zero, then the byte value is used as the RunCount and the next RunCount bytes are read literally from the encoded data stream.

For Further Information

For further information about Microsoft Paint, contact:

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Voice: 206-882-8080
FAX: 206-936-7329
BBS: 206-637-9009
WWW: *http://www.microsoft.com/*

The Microsoft Windows Programmer's Reference Library is the master reference for programmers working with all aspects of Microsoft Windows. The books in this library are supplied with the Microsoft Windows Software Development Kit (SDK). The manuals supplied with the Microsoft C 7.0 Professional Development Systems are also very helpful. You can get information about obtaining these products from:

Microsoft Information Center
Voice: 800-426-9400

Ex.1007 / Page 580 of 1158
APPLE INC.

You may also be able to get information via FTP through the Developer Relations Group at:

ftp://ftp.microsoft.com/developer/drg/

Ex.1007 / Page 581 of 1158
APPLE INC.

Microsoft RIFF

NAME:	Microsoft RIFF
ALSO KNOWN AS:	RIFF, Resource Interchange File Format, RIFX, .WAV, .AVI, .BND, .RMI, .RDI
TYPE:	Multimedia
COLORS:	24-bit
COMPRESSION:	RLE, uncompressed, audio, video
MAXIMUM IMAGE SIZE:	Varies
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little- and big-endian
ORIGINATOR:	Microsoft Corporation
PLATFORM:	Microsoft Windows 3.x, Windows NT
SUPPORTING APPLICATIONS:	Microsoft Windows and OS/2 multimedia applications
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	IFF, Chapter 10, *Multimedia*

USAGE: RIFF is a device control interface and common file format native to the Microsoft Windows system. It is used to store audio, video, and graphics information used in multimedia applications.

COMMENTS: A complex format designed to accommodate various types of data for multimedia applications. Because it is quite new and vendor-controlled, the specification is likely to change in the future.

Overview

Microsoft RIFF (Resource Interchange File Format) is a multimedia file format created by Microsoft for use with the Windows GUI. RIFF itself does not define any new methods of storing data, as many of the bitmap formats described in this book do. Instead, RIFF defines a structured framework, which may contain existing data formats. Using this concept, you can create new, composite formats consisting of two or more existing file formats.

Multimedia applications require the storage and management of a wide variety of data, including bitmaps, audio data, video data, and peripheral device

Ex.1007 / Page 582 of 1158
APPLE INC.

control information. RIFF provides an excellent way to store all these varied types of data. The type of data a RIFF file contains is indicated by the file extension. Examples of data that may be stored in RIFF files are:

- Audio/visual interleaved data (.AVI)
- Waveform data (.WAV)
- Bitmapped data (.RDI)
- MIDI information (.RMI)
- A bundle of other RIFF files (.BND)

NOTE

At this point, AVI files are the only type of RIFF files that have been fully implemented using the current RIFF specification. Although WAV files have been implemented, these files are very simple, and their developers typically use an older specification in constructing them.

Because RIFF is an umbrella name for a variety of multimedia files, RIFF files are referred to by the type of data they contain, rather than by the actual format name of RIFF. For this reason, you may find RIFF files rather confusing when you start to use them. For example, a RIFF file containing Audio/Visual Interleaved data is normally referred to simply as an "AVI file" and not as a "RIFF Audio/Visual Interleaved Format File." Only a programmer might ever realize that all of these different files are the same format, or even care.

There is another area of potential confusion. Some people think that RIFF files are somehow similar in design to TIFF (Tag Image File Format) files. While it is true that both formats contain data structures that may be added or deleted to a file ("tags" in TIFF and "chunks" in RIFF), the internal concept and design of these structures within RIFF and TIFF differ greatly. Unlike TIFF, the RIFF file format is based on the Electronic Arts Interchange File Format (IFF) structure (see the article describing this format). And, although both formats use the same concept of data storage, they are not compatible in their design.

File Organization

RIFF is a binary file format containing multiple nested data structures. Each data structure within a RIFF file is called a *chunk*. Chunks do not have fixed positions within a RIFF file, and therefore standard offset values cannot be used to locate their fields. A chunk contains data such as a data structure, a data

Ex.1007 / Page 583 of 1158
APPLE INC.

stream, or another chunk called a *subchunk*. Every RIFF chunk has the following basic structure:

```
typedef struct _Chunk
{
    DWORD ChunkId;              /* Chunk ID marker */
    DWORD ChunkSize;            /* Size of the chunk data in bytes */
    BYTE ChunkData[ChunkSize];  /* The chunk data */

} CHUNK;
```

ChunkId contains four ASCII characters that identify the data the chunk contains. For example, the characters RIFF are used to identify chunks containing RIFF data. If an ID is smaller than four characters, it is padded on the right using spaces (ASCII 32). Note that RIFF files are written in little-endian byte order. Files written using the big-endian byte ordering scheme have the identifier RIFX.

ChunkSize is the length of the data stored in the ChunkData field, not including any padding added to the data. The size of the ChunkId and ChunkSize fields are not themselves included in this value.

ChunkData contains data that is WORD-aligned within the RIFF file. If the data is an odd length in size, an extra byte of NULL padding is added to the end of the data. The ChunkSize value does not include the length of the padding.

Subchunks also have the same structure as chunks. A subchunk is simply any chunk that is contained within another chunk. The only chunks that may contain subchunks are the RIFF file chunk RIFF and the list chunk, LIST (explained in the next section). All other chunks may contain only data.

A RIFF file itself is one entire RIFF chunk. All other chunks and subchunks in the file are contained within this chunk. If you are decoding, your RIFF reader should ignore any chunks that the reader does not recognize or it cannot use. If you are encoding, your RIFF writer will write out all unknown and unused chunks that were read. Do not discard them.

File Details

RIFF files that are used to store audio and video information are called AVI files. The RIFF AVI file format normally contains only a single AVI chunk; however, other types of chunks may also appear. An AVI reader should ignore all chunks it does not need or recognize that are stored within a RIFF AVI file.

Ex.1007 / Page 584 of 1158
APPLE INC.

Although Microsoft uses a standard notation to describe the internal arrangement of data structures within RIFF files, we believe it is clearer to use our own C-like syntax to illustrate the placement of chunks and subchunks within a RIFF AVI file. The ChunkId for each chunk is listed in the comments:

```
struct _RIFF   /* "RIFF" */
{
    struct _AVICHUNK   /* "AVI " */
    {
        struct _LISTHEADERCHUNK   /* "hdrl" */
        {
            AVIHEADER AviHeader;      /* "avih" */
            struct _LISTHEADERCHUNK  /* "strl" */
            {
                AVISTREAMHEADER       StreamHeader; /* "strh" */
                AVISTREAMFORMAT       StreamFormat; /* "strf" */
                AVISTREAMDATA StreamData;   /* "strd" */
            }
        }
        struct _LISTMOVIECHUNK  /* "movi" */
        {
            struct _LISTRECORDCHUNK  /* "rec " */
            {
                /* Subchunk 1 */
                /* Subchunk 2 */
                /* Subchunk N */
            }
        }
        struct _AVIINDEXCHUNK  /* "idx1" */
        {
            /* Index data */
        }
    }
}
```

The above structure represents the internal data layout of a RIFF file containing only one AVI chunk. This chunk follows the format of the chunk data structure previously described. The AVI chunk is identified by the 4-character chunk identifier "AVI " (note the final blank character). The AVI chunk contains two mandatory LIST subchunks, which indicate the format of the data stream(s) stored in the file.

AVI Header Subchunk

The first mandatory LIST chunk contains the main AVI header subchunk and has the identifier hdrl. The information in the header subchunk defines the format of the entire AVI chunk. The hdrl chunk must appear as the first chunk within the AVI chunk. The format of the header subchunk is the following:

Ex.1007 / Page 585 of 1158
APPLE INC.

```
typedef struct _AVIHeader
{
    DWORD TimeBetweenFrames;      /* Time delay between frames */
    DWORD MaximumDataRate;        /* Data rate of AVI data */
    DWORD PaddingGranularity;     /* Size of single unit of padding */
    DWORD Flags;                  /* Data parameters */
    DWORD TotalNumberOfFrames;    /* Number of video frame stored */
    DWORD NumberOfInitialFrames; /* Number of preview frames */
    DWORD NumberOfStreams;        /* Number of data streams in chunk*/
    DWORD SuggestedBufferSize;    /* Minimum playback buffer size */
    DWORD Width;                  /* Width of video frame in pixels */
    DWORD Height;                 /* Height of video frame in pixels*/
    DWORD TimeScale;              /* Unit used to measure time */
    DWORD DataRate;               /* Data rate of playback */
    DWORD StartTime;              /* Starting time of AVI data */
    DWORD DataLength;             /* Size of AVI data chunk */

} AVIHEADER;
```

TimeBetweenFrames contains a value indicating the amount of delay between frames in microseconds.

MaximumDataRate value indicates the data rate of the AVI data in bytes per second.

PaddingGranularity specifies the multiple size of padding used in the data in bytes. When used, the value of this field is typically 2048.

Flags contains parameter settings specific to the AVI file and its data. The parameters correspond to the bit values of the Flags field as follows:

Bit 4 AVI chunk contains an index subchunk (idx1).

Bit 5 Use the index data to determine how to read the AVI data, rather than the physical order of the chunks with the RIFF file.

Bit 8 AVI file is interleaved.

Bit 16 AVI file is optimized for live video capture.

Bit 17 AVI file contains copyrighted data.

TotalNumberOfFrames indicates the total number of frames of video data stored in the movi subchunk.

NumberOfInitialFrames specifies the number of frames in the file before the actual AVI data. For non-interleaved data this value is 0.

Ex.1007 / Page 586 of 1158
APPLE INC.

NumberOfStreams holds the number of data streams in the chunk. A file with an audio and video stream contains a value of 2 in this field, while an AVI file containing only video data has 1. In the current version of the RIFF format, one audio and one video stream are allowed.

SuggestedBufferSize is the minimum size of the buffer to allocate for playback of the AVI data. For non-interleaved AVI data, this value is at least the size of the largest chunk in the file. For interleaved AVI files, this value should be the size of an entire AVI record.

Width and Height values indicate the size of the video image in pixels.

TimeScale is the unit used to measure time in this chunk. It is used with DataRate to specify the time scale that the stream will use. For video streams, this value should be the frame rate and typically has a value of 30. For audio streams, this value is typically the audio sample rate.

DataRate is divided by the TimeScale value to calculate the number of samples per second.

StartTime is the starting time of the AVI data and is usually 0.

DataLength is the size of the AVI chunk in the units specified by the TimeScale value.

The hdrl subchunk also contains one or more LIST chunks with the identifier strl. There will be one of these LIST chunks per data stream stored in the AVI chunk.

Three subchunks are stored within the strl LIST chunk. The first is the Stream Header subchunk, which has the identifier strh. This header contains information specific to the data stream stored in the strl LIST chunk. A stream header is required and has the following format:

```
typedef struct _StreamHeader
{
    char  DataType[4];          /* Chunk identifier ("strl") */
    char  DataHandler[4];       /* Device handler identifier */
    DWORD Flags;                /* Data parameters */
    DWORD Priority;             /* Set to 0 */
    DWORD InitialFrames;        /* Number of initial audio frames */
    DWORD TimeScale;            /* Unit used to measure time */
    DWORD DataRate;             /* Data rate of playback */
    DWORD StartTime;            /* Starting time of AVI data */
    DWORD DataLength;           /* Size of AVI data chunk */
    DWORD SuggestedBufferSize;  /* Minimum playback buffer size */
```


Ex.1007 / Page 587 of 1158
APPLE INC.

```
    DWORD Quality;              /* Sample quailty factor */
    DWORD SampleSize;           /* Size of the sample in bytes */

} STREAMHEADER;
```

DataType contains a 4-character identifier indicating the type of data the stream header refers to. Identifiers supported by the current version of the RIFF format are: vids for video data and auds for audio data.

DataHandler may contain a 4-character identifier specifying the preferred type of device to handle the data stream.

Flags contains a set of bit flags use to indicate parameter settings related to the data.

Priority is set to 0.

InitialFrames indicates in seconds how far the audio is placed ahead of the video in interleaved data.

TimeScale, DataRate, StartTime, DataLength, and SuggestedBufferSize all have the same function as the fields of the same names in the hdrl chunk.

Quality is an integer in the range of 0 to 10,000, indicating the quality factor used to encode the sample.

SampleSize is the size of a single sample of data. If this value is 0, the sample varies in size and each sample is stored in a separate subchunk. If this value is non-zero, then all the samples are the same size and are stored in a single subchunk.

Immediately following the stream header is a stream format subchunk with the identifier strf. This header describes the format of the stream data. Its format varies depending on the type of data that is stored (audio or video). This subchunk is also required.

Another stream data subchunk with the identifier strd can optionally follow the stream format subchunk. The data in this chunk is used to configure the drivers required to interpret the data. The format of this chunk also varies depending upon the type of compression used on the stream data.

AVI Data Subchunk

The second mandatory LIST chunk contains the actual AVI data, has the identifier movi, and must appear as the second chunk within the AVI chunk.

Ex.1007 / Page 588 of 1158
APPLE INC.

The data in the movi chunk may be grouped in the form of LIST records (a LIST chunk containing one or more subchunks each with the identifier "rec "). Only data that is interleaved to be read from a CD-ROM is stored as a series of LIST records (data is read more efficiently from a CD-ROM when it is interleaved). If the data is not interleaved, it is stored as a single block of data within the movi chunk itself.

Index Chunk

The AVI chunk may also contain a third chunk, called an index chunk. An index chunk has the identifier idx1 and must appear after the hdrl and movi chunks. This chunk contains a list of all chunks within the AVI chunk, along with their locations, and is used for random access of audio and video data. The index chunk has the following format:

```
typedef struct _AviIndex
{
    DWORD Identifier;     /* Chunk identifier reference */
    DWORD Flags;          /* Type of chunk referenced */
    DWORD Offset;         /* Position of chunk in file */
    DWORD Length;         /* Length of chunk in bytes */

} AVIINDEX;
```

Identifier contains the 4-byte identifier of the chunk it references (strh, strf, strd, and so on).

Flags bits are used to indicate the type of frame the chunk contains or to identify the index structure as pointing to a LIST chunk.

Offset indicates the start of the chunk in bytes relative to the movi list chunk.

Length is the size of the chunk in bytes.

The idx1 chunk contains one of these structures for every chunk and subchunk in the AVI chunk. The structures need not index each chunk in the order in which they occur within the AVI chunk. The order of the index structures in the idx1 may also be used to control the presentation order of the data stored in the AVI chunk. If an index is included in an AVI chunk, the appropriate indication bit must be set in the Flags field of the AVI header chunk. If an application reading a RIFF file decides to use the information in the index chunk, it must first find the hdrl chunk and determine if an index chunk exists by examining the Flags field value in the AVI header. If it does exist, the reader will skip past all the chunks in the AVI chunk until it encounters the idx1 chunk.

Ex.1007 / Page 589 of 1158
APPLE INC.

JUNK Chunk

One other type of chunk that is commonly encountered in an AVI chunk is the padding or JUNK chunk (so named because its chunk identifier is JUNK). This chunk is used to pad data out to specific boundaries (for example, CD-ROMs use 2048-byte boundaries). The size of the chunk is the number of bytes of padding it contains. If you are reading AVI data, do not use use the data in the JUNK chunk. Skip it when reading and preserve it when writing. The JUNK chunk uses the standard chunk structure:

```
typedef struct _JunkChunk
{
    DWORD ChunkId;              /* Chunk ID marker (JUNK)*/
    DWORD PaggingSize;          /* Size of the padding in bytes */
    BYTE Padding[ChunkSize];    /* Padding */

} JUNKCHUNK;
```

For Further Information

For further information about the Microsoft RIFF format, see the specification included on the CD-ROM that accompanies this book.

If you write an application that recognizes the RIFF file format, you will need to get a copy of the Microsoft Multimedia Development Kit (MDK). The MDK contains all the tools and documentation necessary to work with RIFF files, as well as with the other details of Microsoft Windows multimedia.

For information about Microsoft multimedia products, including the MDK, contact Microsoft:

Microsoft Corporation
Attn: Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399
WWW: *http://www.microsoft.com/*

For specific information about Microsoft AVI and the RIFF file formats, see the following Microsoft documents:

Microsoft Corporation. *Microsoft Windows Multimedia Programmer's Guide,* Microsoft Press, Redmond, WA.

Microsoft Corporation. *Microsoft Windows Multimedia Programmer's Reference,* Microsoft Press, Redmond, WA.

Ex.1007 / Page 590 of 1158
APPLE INC.

See also the discussion and additional references in Chapter 10, in this book.

You may also be able to get information via FTP through the Developer Relations Group at:

ftp://ftp.microsoft.com/developer/drg/

Ex.1007 / Page 591 of 1158
APPLE INC.

Microsoft RTF

NAME:	Microsoft RTF
ALSO KNOWN AS:	Rich Text Format
TYPE:	Metafile
COLORS:	256
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Microsoft Corporation
PLATFORM:	MS-DOS
SUPPORTING APPLICATIONS:	Most word processing, some spreadsheet
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	None

USAGE: Used for document data interchange.

COMMENTS: A least-common-denominator format used mainly in word-processor documents.

Overview

Microsoft RTF (Rich Text Format) is a metafile standard developed by Microsoft Corporation to encode formatted text and graphics for interchange between applications. Normally, exporting a formatted file from one word processor to another requires that the file be converted from its original format to the format supported by the target application. This conversion almost never produces a target document that is an exact functional duplicate of the original. This is due both to the different features present in the word processor formats, and to limitations of the format converters. If a document is stored as an RTF file, however, and the reading application can also handle RTF files, no intermediate conversion is necessary and therefore no data is misinterpreted or lost.

Ex.1007 / Page 592 of 1158
APPLE INC.

RTF has excellent font-handling capabilities and bitmap storage features. RTF files contain only 7-bit ASCII characters, so the format can support documents formatted using the ANSI, MS-DOS, and Macintosh character sets. These features and others make the RTF format a good choice for use as a multiplatform interchange format.

File Organization

The encoded data in RTF files is arranged more like a stream than a fixed data structure, so there is no definite information header that is the same in all RTF files. Instead, an RTF code stream consists of variable-sized fields called *control words*, *control symbols*, and *groups*. Each of these three types of fields begins with a backslash character (\), followed by one or more ASCII characters. A control word is an RTF code that contains special formatting and printing instructions.

File Details

Looking at the 22 lines of RTF code included in this section, we see the following control codes at the beginning of the file:

```
\rtf1\ansi
```

These control codes indicate that this data stream is an RTF document, that the code conforms to version 1 of the RTF specification, and that the document uses the ANSI (\ansi) rather than the PC (\pc), PS/2 (\pca), or Macintosh (\mac) character sets.

Control symbols are special escape character sequences consisting of a backslash that is followed by a single, nonalphabetic character. RTF control symbols include:

```
\~    Nonbreaking space
\_    Nonbreaking hyphen
\:    Index subentry
\'    Hexadecimal value xx
```

A group is a collection of text, control words, and control symbols, enclosed in a set of braces ({}). In fact, the entire RTF code stream is considered a group and is always enclosed in braces. The first control word in the group identifies the group type. Both the backslash (\) and the brace characters ({}) have special meanings in RTF and should be preceded by a backslash if they are to be interpreted as text.

Ex.1007 / Page 593 of 1158
APPLE INC.

```
{\rtf1\ansi \deff0\deflang1024
{\fonttbl{\f0\froman Tms Rmn;}{\f1\froman Symbol;}{\f2\fswiss Helv;}}
{\colortbl;\red0\green0\blue0;\red0\green0\blue255;\red0\green255\blue255;
\red0\green255\blue0;\red255\green0\blue255;\red255\green0\blue0;
\red255\green255\blue0;\red255\green255\blue255;\red0\green0\blue127;
\red0\green127\blue127;\red0\green127\blue0;\red127\green0\blue127;
\red127\green0\blue0;\red127\green127\blue0;\red127\green127\blue127;
\red192\green192\blue192;}
{\stylesheet{\fs20\lang1033 \snext0 Normal;}}
{\info{\author \'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00}
{\operator \'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00\'00}
{\creatim\yr1992\mo1\dy9\hr12\min53}
{\revtim\yr1992\mo1\dy9\hr12\min53}{\version1}{\edmins3}{\nofpages0}
{\nofwords0}{\nofchars0}{\vern16504}}
\paperw12240\paperh15840\margl1800\margr1800\margt1440\margb1440\gutter0
\widowctrl\ftnbj \sectd \linex0\endnhere \pard\plain \fs20\lang1033
Four Basic Principles to Unify Mind and Body.
\par \tab 1. Keep one point.
\par \tab 2. Relax completely.
\par \tab 3. Keep weight underside.
\par \tab 4. Extend Ki.
\par }
```

Looking again at the RTF code in the figure, we can see a number of groups. The first group is obviously the \rtf group, which contains the code for the entire file.

The \fonttbl group contains the descriptions of the fonts used within the document. This document defines Times Roman, Symbol, and Helvetica font sets.

The next group, \colortbl, is a color table used to control screen and printer colors. This file defines a basic palette of 16 colors, with each color channel containing an 8-bit index value in the range of 0 to 255.

The \stylesheet group contains descriptions and definitions of the various styles and formats used in the document. In this example, we can see that Normal is the only style defined in this document.

The \info group contains one or more pieces of information about the documents, such as title, subject, author, version, keywords, and comments. In this example, the author and operator (the person who made the last change to the document) are blank. The remaining fields identify the creation time and last revision time of the document and its application version number.

After the groups, we see a series of control words that define the document, section, and paragraph formats, including the width, height, and margins. Following these control words is the actual text, which is one line of text followed by four lines of tab-indented text.

Ex.1007 / Page 594 of 1158
APPLE INC.

RTF can also handle bitmap images encoded in either a hexadecimal or binary format. The control word \pict always begins a group containing bitmapped data. A \pict group might appear in an RTF code stream as follows:

```
{\pict\wmetafile8\picw23918\pich14552\picwgoal13562\pichgoal8251
\picscalex63\piccaley63
```

The control words are the following:

- Source file type
- Image width and height
- Picture width and height
- Horizontal scaling value
- Vertical scaling value

If the image source is a bitmap (\wbitmap), then the following additional control words may appear:

- Bits per pixel
- Number of pixel planes
- Picture width in bytes

Source images may also be Macintosh PICT files.

Following the \pict group is the actual bitmap data, which is hexadecimal in format by default (as shown in the example below). If the data is in binary format, it is preceded by the \bin control word, followed by the number of bytes of binary data that follow.

```
{\rtf1\ansi \deff0\deflang1024
{\fonttbl{\f0\froman CG Times (WN);}{\f1\fdecor Symbol;}{\f2\fswiss Univers (WN);}}
{\colortbl;\red0\green0\blue0;\red0\green0\blue255;\red0\green255\blue255;
\red0\green255\blue0;\red255\green0\blue255;\red255\green0\blue0;
\red255\green255\blue0;\red255\green255\blue255;\red0\green0\blue127;
\red0\green127\blue127;\red0\green127\blue0;\red127\green0\blue127;
\red127\green0\blue0;\red127\green127\blue0;\red127\green127\blue127;
\red192\green192\blue192;}
{\stylesheet{\fs20\lang1033 snext0 Normal;}}
{\info{\author James D. Murray}
{\creatim\yr1992\mo1\dy9\hr15\min31}{\printim\yr1992\mo1\dy9\hr15\min32}
{\version1}{\edmins2}{\nofpages1}{\nofwords0}{\nofchars2}{\vern16504}}
\paperw12240\paperh15840\margl1800\margr1800\margt1440\margb1440\gutter0
\widowctrl\ftnbj \sectd \linex0\endnhere \pard\plain \fs20\lang1033
{\pict\wmetafile8\picw23918\pich14552\picwgoal13562\pichgoal8251
\picscalex63\picscaley63
01000900000328ea01000000fee901000000050000000b020000000005000000c024c0410070500
```


Ex.1007 / Page 595 of 1158
APPLE INC.

```
00000b020000000005000000c024c041007050000000902000000000500000000102ffffff00fee9
0100430f2000cc0000004c041007000000004c041007000000002800000100700004c0400000100
010000000000000000000000000000000000000000000000000000000000ffffff00ffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
0000}\par}
```

For Further Information

For further information, see the specification included on the CD-ROM that accompanies this book. You may be able to get additional information by contacting Microsoft:

Microsoft Corporation
Attn: Department RTF
16011 N.E. 36th Way
Box 97017
Redmond, WA 98073-9717
WWW: *http://www.microsoft.com/*

The RTF file format is also documented in the following reference:

Microsoft Corporation. *Microsoft Word Technical Reference Manual,* Microsoft Press, Redmond, WA.

This book is available in bookstores or from:

Microsoft Press
Voice: 800-677-7377

You may also be able to get information via FTP through the Developer Relations Group at:

ftp://ftp.microsoft.com/developer/drg/

Ex.1007 / Page 596 of 1158
APPLE INC.

NAME:	Microsoft SYLK
ALSO KNOWN AS:	Symbolic Link Format, SLK
TYPE:	Vector
COLORS:	NA
COMPRESSION:	NA
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	Microsoft Corporation
PLATFORM:	MS-DOS, others
SUPPORTING APPLICATIONS:	Spreadsheets, business graphics applications
SPECIFICATION ON CD:	No
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	Lotus DIF

USAGE: Interchange of spreadsheet information.

COMMENTS: Yet another format used to share spreadsheet information with business graphics applications.

Overview

The Microsoft SYLK (Symbolic Link) format is used mainly for the interchange of spreadsheet data between applications such as Microsoft Multiplan and Excel. Files in this format might also be imported directly by business graphics applications. SYLK files are written entirely in ASCII and, like Lotus DIF and SDI, are application-independent. SYLK, however, incorporates several features not found in other spreadsheet data interchange formats.

File Organization

Records in a SYLK file contain three fields: a Record Type Descriptor (RTD), a

Ex.1007 / Page 597 of 1158
APPLE INC.

Field Type Descriptor (FTD), and a variable amount of data. A SYLK record has the following format:

```
<RTD>;<FTD>;<data> ...
```

File Details

The following Record Type Descriptors (RTDs) are currently defined by SYLK:

RTD	Description
B	Cell boundary
C	A data cell
E	End of file
F	Cell formatting parameter
ID	SYLK file identification record
NE	Link to an inactive spreadsheet file
NN	Name given to a rectangluar area of cells
NU	Substitute filename
P	Time and date stamp formats

Each Record Type Decriptor may be followed by a single Field Type Descriptor (FTD) if needed. Most field type descriptors have meanings unique to each record, but a few, listed below, have meanings global to all record types:

FTD	Description
W	Column width
X	Horizontal cell coordinate
Y	Vertical cell coordinate

The SYLK file format does not contain a header and resembles a data stream in its design. Except for the ID record, which must be the first record in every SYLK file, RTDs may appear anywhere in the file with the following exceptions:

- The first record must be an ID record (the RTD is ID)
- All P records follow the ID record.
- All B records follow the P records.

Ex.1007 / Page 598 of 1158
APPLE INC.

- A ;D or ;G FTD must appear in a C record prior to a reference to that FTD by another record.
- NE records always follow NU records.
- The final record must be an E record.

For Further Information

SYLK was created and is maintained by Microsoft Corporation. You may be able to get information by contacting:

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Voice: 206-882-8080
Voice: 800-426-9400
FAX: 206-883-8101
WWW: *http://www.microsoft.com/*

The following reference also contains information about the SYLK format:

Walden, Jeffrey B. *File Formats for Popular PC Software,* John Wiley & Sons, New York, NY, 1986.

You may also be able to get information via FTP through the Developer Relations Group at:

ftp://ftp.microsoft.com/developer/drg/

Ex.1007 / Page 599 of 1158
APPLE INC.

Microsoft Windows Bitmap

NAME:	Microsoft Windows Bitmap*
ALSO KNOWN AS:	BMP, DIB, Windows BMP, Windows DIB, Compatible Bitmap
TYPE:	Bitmap
COLORS:	1-, 4-, 8-, 16-, 24-, and 32-bits
COMPRESSION:	RLE, uncompressed
MAXIMUM IMAGE SIZE:	32K×32K and 2G×2G pixels
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Microsoft Corporation
PLATFORM:	Intel machines running Microsoft Windows, Windows NT, Windows 95, OS/2, and MS-DOS
SUPPORTING APPLICATIONS:	Too numerous to list
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes
IMAGES ON CD:	Yes
SEE ALSO:	OS/2 Bitmap

USAGE: Used as the standard bitmap storage format in the Microsoft Windows environment. Although it is based on Windows internal bitmap data structures, it is supported by many non-Windows and non-PC applications.

COMMENTS: A well-defined format for programmers having access to the Microsoft Developer's Network Knowledge Base and Software Development Kits (SDKs). Its simple RLE compression scheme is rather inefficient for complex images. Its many variations and differences from the OS/2 BMP format can be confusing.

Overview

The Microsoft Windows Bitmap (BMP) file format is one of several graphics file formats supported by the Microsoft Windows operating environment. BMP is the native bitmap format of Windows and is used to store virtually any type of bitmap data. Most graphics and imaging applications running under Microsoft

* Our thanks to David Charlap for his contributions to this article.

Ex.1007 / Page 600 of 1158
APPLE INC.

Windows support the creation and display of BMP format files. BMP is also very popular in the MS-DOS operating system. It is also the native bitmap file format of OS/2.

The original bitmap format created for Windows 1.0 was very simple. It had a fixed color palette, did not support bitmap data compression, and was designed to support the most popular IBM PC graphics cards in use at the time (CGA, EGA, Hercules, and others). This defunct format is now often referred to as the original Windows device-dependent bitmap (DDB).

As Windows 2.0 was being developed, support for a programmable color palette was added to both Windows and BMP, allowing user-definable color data to be stored along with the bitmap data that used it. When stored in memory, this collection of information is known as a Microsoft Device Independent Bitmap, or DIB. When this information is written out to a file, it is known as the Microsoft Bitmap Format, or BMP. When you hear references to the DIB file format, it is BMP that is actually being referred to.

During the development of BMP, Microsoft shared responsibility with IBM for the development of early versions of IBM's OS/2 operating system. When Presentation Manager, the OS/2 graphical user interface, required a bitmap format, the Windows BMP format was used. Thus, the Windows 2.*x* and OS/2 1.*x* BMP formats are identical.

The BMP format modified for Windows 3.0 differs only slightly from the OS/2 Presentation Manager bitmap format that preceded it. Note that later revisions designed to support IBM OS/2 Presentation Manager 2.*x* have resulted in further divergence between the Microsoft Windows and IBM OS/2 BMP file formats. The current version of BMP for Windows 4.0 (Windows 95) contains all of the features and history of the Windows 2.*x*, 3.*x*, and Windows NT BMP formats.

The structure of BMP format files is closely tied to the API of both Windows and OS/2. In this regard, BMP was never meant to be a portable format or used for bitmap data interchange between different operating systems. As each of these operating system APIs has changed, the BMP format has changed along with it.

There are currently three versions of BMP under Windows (2.*x*, 3.*x*, and 4.*x* [Windows 95]), two versions under OS/2 (1.*x* and 2.*x*, with six possible variations), and a single version for Windows NT. This article details the three versions used under Microsoft Windows, as well as the Windows NT version. The original Microsoft device-dependent bitmap format is also discussed. For a

Ex.1007 / Page 601 of 1158
APPLE INC.

discussion of the OS/2 BMP format versions and variants, see the article about the OS/2 BMP format.

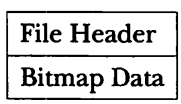
All of the BMP versions originated on Intel-based machines and thus share a common little-endian heritage. The current BMP format is otherwise hardware-independent and can accommodate images with up to 32-bit color. Its basic design makes it a good general purpose format that can be used for color or black-and-white image storage if file size is not a factor. Its main virtues are its simplicity and widespread support in the PC marketplace.

The compression method used is a type of run-length encoding (RLE), although most BMP files to date have been stored uncompressed. A notable exception is the Microsoft Windows 3.1 sign-on screen shipped with all copies of the product. Although the BMP RLE scheme is lossless and easily and quickly decompressed, it is not considered a superior compression method.

Although the BMP format is well-defined, there is no actual format specification document published by Microsoft. Information about structure and data encoding methods is contained in a number of programmer's references, manuals, online help facilities, and include files associated with the Microsoft Windows Software Development Kits (SDKs) and Microsoft Developers Network Knowledge Base.

File Organization

Windows 1.*x* DDB files contain two sections: a file header and the bitmap data. There is no provision for a color palette or any other features that would make this format device-independent. Support for compression of the bitmap data is also lacking.



Windows 2.*x*, 3.*x*, and 4.*x* BMP files contain four sections: a file header, a bitmap information header, a color palette, and the bitmap data. Of these four sections, only the palette information may be optional, depending on the bit depth of the bitmap data. The BMP file header is 14 bytes in length and is nearly identical to the 1.*x* DDB header. The file header is followed by a second

Ex.1007 / Page 602 of 1158
APPLE INC.

header (called the bitmap header), a variable-sized palette, and the bitmap data.

File Header
Bitmap Header
Color Palette
Bitmap Data

File Details

This section describes the original Windows DDB format and BMP format versions 2*x*, 3*x*, and 4*x* in greater detail.

Version 1 Device-Dependent Bitmap (Microsoft Windows 1.x)

DDB files contain only a file header followed by uncompressed bitmap data. The following shows the structure of the 10-byte DDB file header:

```
typedef struct _Win1xHeader
{
    WORD Type;          /* File type identifier (always 0) */
    WORD Width;         /* Width of the bitmap in pixels */
    WORD Height;        /* Height of the bitmap in scan lines */
    WORD ByteWidth;     /* Width of bitmap in bytes */
    BYTE Planes;        /* Number of color planes */
    BYTE BitsPerPixel;  /* Number of bits per pixel */
} WIN1XHEADER;
```

Type indicates the file type; for v1.*x* headers, it is always 0.

Width and Height represent the size of the bitmap in pixels and in scan lines, respectively.

ByteWidth shows the width of the bitmap in bytes. It is assumed that this value will include the size of any scan line padding that is present.

Planes is the number of color planes used to store the bitmap. This value is always 1.

BitsPerPixel is the size of each pixel in bits. This value is typically 1, 4, or 8.

The image data immediately follows the header and is stored in an uncompressed format. Each pixel stores an index value into the fixed system colormap used by Windows 1.0. The presence of scan line padding may be

Ex.1007 / Page 603 of 1158
APPLE INC.

determined by comparing the calculated width of a line in bytes with the actual width of the line in bytes stored as the value of the ByteWidth field.

BMP Version 2 (Microsoft Windows 2.x)

All versions of BMP format files begin with the following 14-byte header:

```
typedef struct _WinBMPFileHeader
{
    WORD   FileType;      /* File type, always 4D42h ("BM") */
    DWORD  FileSize;      /* Size of the file in bytes */
    WORD   Reserved1;     /* Always 0 */
    WORD   Reserved2;     /* Always 0 */
    DWORD  BitmapOffset; /* Starting position of image data in bytes */
} WINBMPFILEHEADER;
```

FileType holds a 2-byte magic value used to identify the file type; it is always 4D42h or "BM" in ASCII. If your application reads Windows bitmap files, make sure to always check this field before attempting to use any of the data read from the file.

FileSize is the total size of the BMP file in bytes and should agree with the file size reported by the filesystem. This field only stores a useful value when the bitmap data is compressed, and this value is usually zero in uncompressed BMP files.

Reserved1 and Reserved2 do not contain useful data and are usually set to zero in a BMP header written to disk. These fields are instead used by an application when the header is read into memory.

BitmapOffset is the starting offset of the bitmap data from the beginning of the file in bytes.

Following the file header in v2.*x* BMP files is a second header called the bitmap header. This header contains information specific to the bitmap data. This header is 12 bytes in length and has the following format:

```
typedef struct _Win2xBitmapHeader
{
    DWORD Size;             /* Size of this header in bytes */
    SHORT Width;            /* Image width in pixels */
    SHORT Height;           /* Image height in pixels */
    WORD  Planes;           /* Number of color planes */
    WORD  BitsPerPixel;     /* Number of bits per pixel */
} WIN2XBITMAPHEADER;
```

Size is the size of the header in bytes. For Windows 2.*x* BMP files, this value is always 12.

Ex.1007 / Page 604 of 1158
APPLE INC.

Width and Height are the width and height of the image in pixels, respectively. If Height is a positive number, then the image is a "bottom-up" bitmap with the origin in the lower-left corner. If Height is a negative number, then the image is a "top-down" bitmap with the origin in the upper-left corner. Width does not include any scan-line boundary padding.

Planes is the number of color planes used to represent the bitmap data. BMP files contain only one color plane, so this value is always 1.

BitsPerPixel is the number of bits per pixel in each plane. This value will be in the range 1 to 24; the values 1, 4, 8, and 24 are the only values considered legal by the Windows 2.*x* API.

The Windows 2.*x* bitmap header is identical to the OS/2 1.*x* bitmap header except that the Width and Height fields are signed values in Windows BMP files.

Following the header is the color palette data. A color palette is always present in a BMP file if the bitmap data contains 1-, 4-, or 8-bit data. Twenty-four-bit bitmap data never uses a color palette (nor does it ever need to). Each element of the palette is three bytes in length and has the following structure:

```
typedef struct _Win2xPaletteElement
{
    BYTE Blue;      /* Blue component */
    BYTE Green;     /* Green component */
    BYTE Red;       /* Red component */
} WIN2XPALETTEELEMENT;
```

Blue, Green, and Red hold the color component values for a pixel; each is in the range 0 to 255.

The size of the color palette is calculated from the BitsPerPixel value. The color palette has 2, 16, 256, or 0 entries for a BitsPerPixel of 1, 4, 8, and 24, respectively. The number of color palette entries is calculated as follows:

```
NumberOfEntries = 1 << BitsPerPixel;
```

To detect the presence of a color palette in a BMP file (rather than just assuming that a color palette does exist), you can calculate the number of bytes between the bitmap header and the bitmap data and divide this number by the size of a single palette element. Assuming that your code is compiled using 1-byte structure element alignment, the calculation is:

```
NumberOfEntries = (BitmapOffset - sizeof(WINBMPFILEHEADER) -
     sizeof(WIN2XBITMAPHEADER)) / sizeof(WIN2XPALETTEELEMENT);
```


Ex.1007 / Page 605 of 1158
APPLE INC.

If NumberOfEntries is zero, there is no palette data; otherwise, you have the number of elements in the color palette.

BMP Version 3 (Microsoft Windows 3.x)

Version 3.*x* BMP files begin with the same 14-byte header as v2.*x* BMP files. The file header is also followed by a bitmap header, which is an expanded version of the v2.*x* bitmap header. It is 40 bytes in size and contains six additional fields:

```
typedef struct _Win3xBitmapHeader
{
    DWORD Size;             /* Size of this header in bytes */
    LONG  Width;            /* Image width in pixels */
    LONG  Height;           /* Image height in pixels */
    WORD  Planes;           /* Number of color planes */
    WORD  BitsPerPixel;     /* Number of bits per pixel */

    /* Fields added for Windows 3.x follow this line */

    DWORD Compression;      /* Compression methods used */
    DWORD SizeOfBitmap;     /* Size of bitmap in bytes */
    LONG  HorzResolution;   /* Horizontal resolution in pixels per meter */
    LONG  VertResolution;   /* Vertical resolution in pixels per meter */
    DWORD ColorsUsed;       /* Number of colors in the image */
    DWORD ColorsImportant; /* Minimum number of important colors */
} WIN3XBITMAPHEADER;
```

Size is the size of the header in bytes. For Windows 3.*x* BMP files, this value is always 40.

Width and Height are the width and height of the image in pixels, respectively. If Height is a positive number, then the image is a "bottom-up" bitmap with the origin in the lower-left corner. If Height is a negative number, then the image is a "top-down" bitmap with the origin in the upper-left corner. Width does not include any scan-line boundary padding.

Planes is the number of color planes used to represent the bitmap data. BMP files contain only one color plane, so this value is always 1.

BitsPerPixel is the number of bits in each pixel. This value is in the range 1 to 24; the values 1, 4, 8, and 24 are the only values considered legal by the Windows 3.*x* API.

Compression indicates the type of encoding method used to compress the bitmap data. 0 indicates that the data is uncompressed; 1 indicates that the 8-bit RLE algorithm was used; 2 indicates that the 4-bit RLE algorithm was used.

Ex.1007 / Page 606 of 1158
APPLE INC.

(See the section called "Image Data and Compression" below for more information on BMP RLE encoding.)

SizeOfBitmap is the size of the stored bitmap in bytes. This value is typically zero when the bitmap data is uncompressed; in this case, the decoder computes the size from the image dimensions.

HorzResolution and VertResolution are the horizontal and vertical resolutions of the bitmap in pixels per meter. These values are used to help a BMP reader choose a proper resolution when printing or displaying a BMP file.

ColorsUsed is the number of colors present in the palette. If this value is zero, and the value of BitsPerPixel is less than 16, then the number of entries is equal to the maximum size possible for the colormap. BMP files with a BitsPerPixel value of 16 or greater will not have a color palette. This value is calculated by using the value of the BitsPerPixel field:

```
ColorsUsed = 1 << BitsPerPixel;
```

ColorsImportant is the number of significant colors in the palette, determined by their frequency of appearance in the bitmap data; the more frequent the occurrence of a color, the more important it is. This field is used to provide as accurate a display as possible when using graphics hardware supporting fewer colors than are defined in the image. For example, an 8-bit image with 142 colors might only have a dozen or so colors making up the bulk of the image. If these colors could be identified, a display adapter with only 16-color capability would be able to display the image more accurately using the 16 most frequently occurring colors in the image. The most important colors are always stored first in the palette; ColorsImportant is 0 if all of the colors in the palette are to be considered important.

The color palette that may follow the bitmap header is basically the same as the v2.*x* palette but adds an extra byte of padding to increase its size to four bytes. This allows palette entries to be read as 4-byte values, making these values more efficient to read in memory and easier to see in a hex dump or debugger.

```
typedef struct _Win3xPaletteElement
{
    BYTE Blue;      /* Blue component */
    BYTE Green;     /* Green component */
    BYTE Red;       /* Red component */
    BYTE Reserved;  /* Padding (always 0) */
} WIN3XPALETTEELEMENT;
```


Ex.1007 / Page 607 of 1158
APPLE INC.

Blue, Green, and Red hold the color component values for a pixel; each is in the range 0 to 255.

Reserved pads the structure to end on an even-byte boundary and is always zero.

BMP Version 3 (Microsoft Windows NT)

Windows NT uses a variation of the Windows 3.*x* BMP format to store 16- and 32-bit data in a BMP file. This variation adds three additional fields that follow the bitmap header in place of a color palette. The bitmap header is 40 bytes in length and has the following format:

```
typedef struct _WinNtBitmapHeader
{
    DWORD Size;            /* Size of this header in bytes */
    LONG  Width;           /* Image width in pixels */
    LONG  Height;          /* Image height in pixels */
    WORD  Planes;          /* Number of color planes */
    WORD  BitsPerPixel;    /* Number of bits per pixel */
    DWORD Compression;     /* Compression methods used */
    DWORD SizeOfBitmap;    /* Size of bitmap in bytes */
    LONG  HorzResolution;  /* Horizontal resolution in pixels per meter */
    LONG  VertResolution;  /* Vertical resolution in pixels per meter */
    DWORD ColorsUsed;      /* Number of colors in the image */
    DWORD ColorsImportant; /* Minimum number of important colors */
} WINNTBITMAPHEADER;
```

All fields are the same as in the v3.*x* BMP format, except for the Compression field.

Compression indicates the type of encoding method used to compress the bitmap data. 0 indicates that the data is uncompressed; 1 indicates that the 8-bit RLE algorithm was used; 2 indicates that the 4-bit RLE algorithm was used; and 3 indicates that bitfields encoding was used. If the bitmap contains 16 or 32 bits per pixel, then only a Compression value of 3 is supported and the RedMask, GreenMask, and BlueMask fields will be present following the header in place of a color palette. If Compression is a value other than 3, then the file is identical to a Windows 3.*x* BMP file.

```
typedef _WinNtBitfieldsMasks
{
    DWORD RedMask;         /* Mask identifying bits of red component */
    DWORD GreenMask;       /* Mask identifying bits of green component */
    DWORD BlueMask;        /* Mask identifying bits of blue component */
} WINNTBITFIELDSMASKS;
```


Ex.1007 / Page 608 of 1158
APPLE INC.

RedMask, GreenMask, and BlueMask specify which bits in a pixel value correspond to a specific color in 16- and 32-bit bitmaps. The bits in these mask values must be contiguous and must not contain overlapping fields. The bits in the pixel are ordered from most significant to least significant bits. For 16-bit bitmaps, the RGB565 format is often used to specify five bits each of red and blue values, and six bits of green:

```
RedMask   = 0xF8000000;     /* 1111 1000 0000 0000 0000 0000 0000 0000 */
GreenMask = 0x07E00000;     /* 0000 0111 1110 0000 0000 0000 0000 0000 */
BlueMask  = 0x001F0000;     /* 0000 0000 0001 1111 0000 0000 0000 0000 */
```

For 32-bit bitmaps, the RGB101010 format can be used to specify 10 bits each of red, green, and blue:

```
RedMask   = 0xFFC00000;     /* 1111 1111 1100 0000 0000 0000 0000 0000 */
GreenMask = 0x003FF000;     /* 0000 0000 0011 1111 1111 0000 0000 0000 */
BlueMask  = 0x00000FFC;     /* 0000 0000 0000 0000 0000 1111 1111 1100 */
```

BMP Version 4 (Microsoft Windows 95)

Version 4.*x* BMP files begin with the same 14-byte header as v2.*x* and v3.*x* BMP files. The file header is also followed by a bitmap header, which is an expanded version of the v3.*x* bitmap header, incorporating the mask fields of the NT BMP format. This v4.*x* bitmap header is 108-bytes in size and contains 17 additional fields:

```
typedef struct _Win4xBitmapHeader
{
    DWORD Size;            /* Size of this header in bytes */
    LONG  Width;           /* Image width in pixels */
    LONG  Height;          /* Image height in pixels */
    WORD  Planes;          /* Number of color planes */
    WORD  BitsPerPixel;    /* Number of bits per pixel */
    DWORD Compression;     /* Compression methods used */
    DWORD SizeOfBitmap;    /* Size of bitmap in bytes */
    LONG  HorzResolution;  /* Horizontal resolution in pixels per meter */
    LONG  VertResolution;  /* Vertical resolution in pixels per meter */
    DWORD ColorsUsed;      /* Number of colors in the image */
    DWORD ColorsImportant; /* Minimum number of important colors */

    /* Fields added for Windows 4.x follow this line */

    DWORD RedMask;         /* Mask identifying bits of red component */
    DWORD GreenMask;       /* Mask identifying bits of green component */
    DWORD BlueMask;        /* Mask identifying bits of blue component */
    DWORD AlphaMask;       /* Mask identifying bits of alpha component */
    DWORD CSType;          /* Color space type */
    LONG  RedX;            /* X coordinate of red endpoint */
```


Ex.1007 / Page 609 of 1158
APPLE INC.

```
    LONG  RedY;          /* Y coordinate of red endpoint */
    LONG  RedZ;          /* Z coordinate of red endpoint */
    LONG  GreenX;        /* X coordinate of green endpoint */
    LONG  GreenY;        /* Y coordinate of green endpoint */
    LONG  GreenZ;        /* Z coordinate of green endpoint */
    LONG  BlueX;         /* X coordinate of blue endpoint */
    LONG  BlueY;         /* Y coordinate of blue endpoint */
    LONG  BlueZ;         /* Z coordinate of blue endpoint */
    DWORD GammaRed;      /* Gamma red coordinate scale value */
    DWORD GammaGreen;    /* Gamma green coordinate scale value */
    DWORD GammaBlue;     /* Gamma blue coordinate scale value */
} WIN4XBITMAPHEADER;
```

Size is the size of the header in bytes. For Windows 4.*x* BMP files, this value is always 108.

Width and Height are the width and height of the image in pixels, respectively. If Height is a positive number, then the image is a "bottom-up" bitmap with the origin in the lower-left corner. If Height is a negative number, then the image is a "top-down" bitmap with the origin in the upper-left corner. Width does not include any scan-line boundary padding.

Planes is the number of color planes used to represent the bitmap data. BMP files contain only one color plane, so this value is always 1.

BitsPerPixel is the number of bits in each pixel. This value is in the range 1 to 24; the values 1, 4, 8, 16, 24, and 32 are the only values considered legal by the Windows 4.*x* API.

Compression indicates the type of encoding method used to compress the bitmap data. 0 indicates that the data is uncompressed; 1 indicates that the 8-bit RLE algorithm was used; 2 indicates that the 4-bit RLE algorithm was used; and 3 indicates that bitfields encoding was used. If the bitmap contains a 16- or 32-bit bitmap, then only a compression value of 3 is supported.

SizeOfBitmap is the size of the stored bitmap in bytes. This value is typically zero when the bitmap data is uncompressed (including bitfields-encoded bitmaps); in this case, the decoder computes the size from the image dimensions.

HorzResolution and VertResolution are the horizontal and vertical resolutions of the bitmap in pixels per meter. These values are used to help a BMP reader choose a proper resolution when printing or displaying a BMP file.

ColorsUsed is the number of colors present in the palette. If this value is zero and the BMP file contains a color palette, then the number of entries is equal to the maximum size possible for the color palette. If the bitmap has a pixel

Ex.1007 / Page 610 of 1158
APPLE INC.

depth of 16 or greater, there is never a color palette, and this value will be zero.

ColorsImportant is the number of significant colors in the palette, determined by their frequency of appearance in the bitmap data; the more frequent the occurrence of a color, the more important it is. See the explanation of this field for the Windows 3.*x* bitmap header for more information.

RedMask, GreenMask, BlueMask, and AlphaMask specify which bits in a pixel value correspond to a specific color or alpha channel in 16- and 32-bit bitmaps. The bits in these mask values must be contiguous and must not contain overlapping fields. The bits in the pixel are ordered from most significant to least significant bits. For example, a 16-bit bitmap using the RGB555 format would specify five bits each of red, green, blue, and alpha as follows:

```
AlphaMask = 0xF8000000;     /* 1111 1000 0000 0000 0000 0000 0000 0000 */
RedMask   = 0x07C00000;     /* 0000 0111 1100 0000 0000 0000 0000 0000 */
GreenMask = 0x003E0000;     /* 0000 0000 0011 1110 0000 0000 0000 0000 */
BlueMask  = 0x0001F000;     /* 0000 0000 0000 0001 1111 0000 0000 0000 */
```

A 32-bit bitmap using the RGB888 format would specify eight bits each of red, green, and blue using the mask values as follows:

```
AlphaMask = 0xFF000000;     /* 1111 1111 0000 0000 0000 0000 0000 0000 */
RedMask   = 0x00FF0000;     /* 0000 0000 1111 1111 0000 0000 0000 0000 */
GreenMask = 0x0000FF00;     /* 0000 0000 0000 0000 1111 1111 0000 0000 */
BlueMask  = 0x000000FF;     /* 0000 0000 0000 0000 0000 0000 1111 1111 */
```

Note that Windows 95 only supports the RGB555 and RGB565 masks for 16-bit BMP bitmaps and RGB888 for 32-bit BMP bitmaps.

CSType is the color space type used by the bitmap data. Values for this field include 00h (calibrated RGB), 01h (device-dependent RGB), and 02h (device-dependent CMYK). Device-dependent RGB is the default color space. Calibrated RGB is defined by the 1931 CIE XYZ standard.

RedX, RedY, and RedZ specify the CIE X, Y, and Z coordinates, respectively, for the endpoint of the red component of a specified logical color space. These fields are used only when CSType is 00h (calibrated RGB).

GreenX, GreenY, and GreenZ specify the CIE X, Y, and Z coordinates, respectively, for the endpoint of the green component of a specified logical color space. These fields are used only when CSType is 00h (calibrated RGB).

BlueX, BlueY, and BlueZ specify the CIE X, Y, and Z coordinates, respectively, for the endpoint of the blue component of a specified logical color space. These fields are used only when CSType is 00h (calibrated RGB).

Ex.1007 / Page 611 of 1158
APPLE INC.

GammaRed, GammaGreen, and GammaBlue are the red, green, and blue gamma coordinate scale values, respectively, for this bitmap.

All of the additional fields added to the Windows 4.*x* bitmap header are used to support 16- and 32-bit bitmaps and color matching and color characterization of the bitmap data. Color processing may be performed on an image and the ICM (Image Color Matching) information stored in the BMP file. This data is used to provide color matching processing when the bitmap is printed or displayed.

Color Palette

As we have seen, a BMP color palette is an array of structures that specify the red, green, and blue intensity values of each color in a display device's color palette. Each pixel in the bitmap data stores a single value used as an index into the color palette. The color information stored in the element at that index specifies the color of that pixel.

One-, 4-, and 8-bit BMP files are expected to always contain a color palette. Sixteen-, 24-, and 32-bit BMP files never contain color palettes. Sixteen- and 32-bit BMP files contain bitfields mask values in place of the color palette.

You must be sure to check the Size field of the bitmap header to know if you are reading 3-byte or 4-byte color palette elements. A Size value of 12 indicates a Windows 2.*x* (or possibly an OS/2 1.*x*) BMP file with 3-byte elements. Larger numbers (such as 40 and 108) indicate later versions of BMP, which all use 4-byte color palette elements.

Windows BMP File Types

Each new version of BMP has added new information to the bitmap header. In some cases, the newer versions have changed the size of the color palette and added features to the format itself. Unfortunately, a field wasn't included in the header to easily indicate the version of the file's format or the type of operating system that created the BMP file. If we add Windows' four versions of BMP to OS/2's two versions—each with four possible variations—we find that as many as twelve different related file formats all have the file extension ".BMP".

It is clear that you cannot know the internal format of a BMP file based on the file extension alone. But, fortunately, you can use a short algorithm to determine the internal format of BMP files.

The FileType field of the file header is where we start. If these two byte values are 424Dh ("BM"), then you have a single-image BMP file that may have been

Ex.1007 / Page 612 of 1158
APPLE INC.

created under Windows or OS/2. If FileType is the value 4142h ("BA"), then you have an OS/2 bitmap array file. Other OS/2 BMP variations have the file extensions .ICO and .PTR.

If your file type is "BM", then you must now read the Size field of the bitmap header to determine the version of the file. Size will be 12 for Windows 2.*x* BMP and OS/2 1.*x* BMP, 40 for Windows 3.*x* and Windows NT BMP, 12 to 64 for OS/2 2.*x* BMP, and 108 for Windows 4.*x* BMP. A Windows NT BMP file will always have a Compression value of 3; otherwise, it is read as a Windows 3.*x* BMP file.

Note that the only difference between Windows 2.*x* BMP and OS/2 1.*x* BMP is the data type of the Width and Height fields. For Windows 2.*x*, they are signed shorts and for OS/2 1.*x*, they are unsigned shorts. Windows 3.*x*, Windows NT, and OS/2 2.*x* BMP files only vary in the number of fields in the bitmap header and in the interpretation of the Compression field.

Image Data and Compression

Uncompressed data is a series of values representing either color palette indices or actual RGB color values. Pixels are packed into bytes and arranged as scan lines. Each scan line must end on a 4-byte boundary, so one, two, or three bytes of padding may follow each scan line.

Scan lines are stored from the bottom up if the value of the Height field in the bitmap header is a positive value; they are stored from the top down if the Height field value is negative. The bottom-up configuration is the most common, because scan lines stored from the top down cannot be compressed.

Monochrome bitmaps contain one bit per pixel, eight pixels per byte (with the most significant bit being the leftmost pixel), and have a 2-element color palette. If a BMP reader chooses to ignore the color palette, all "one" bits are set to the display's foreground color and all "zero" bits are set to the background color.

Four-bit pixels are packed two per byte with the most significant nibble being the leftmost pixel. Eight-bit pixels are stored one per byte. Both 4- and 8-bit pixel values are indices into color palettes 16 and 256 elements in size respectively.

Sixteen-bit pixels in the Windows NT format are two bytes in size and are stored in big-endian order. In other words, on little-endian machines these bytes must be read and flipped into little-endian order before they are used. The organization of the bit fields in the 16-bit pixels is defined by the values of

Ex.1007 / Page 613 of 1158
APPLE INC.

the RedMask, GreenMask, and BlueMask fields in the header. The most common masks are RGB555 and RGB565. The Compression field must always be a value of 3 (bitfields encoding) when a file stores 16-bit data.

In the v4.*x* BMP format, 16- and 32-bit pixels are stored as little-endian 4-byte RGB values. Common masks for 32-bit data include RGB888 and RGB101010. These bit depths also require bitfields encoding and the mask fields in the bitmap header to define their pixel format. 24-bit bitmap data is always stored as 3-byte RGB values.

The Windows BMP format supports a simple run-length encoded (RLE) compression scheme for compressing 4-bit and 8-bit bitmap data. Since this is a byte-wise RLE scheme, 1-, 16-, 24-, and 32-bit bitmaps cannot be compressed using it, due to the typical lack of long runs of bytes with identical values in these types of data.

BMP uses a two-value RLE scheme. The first value contains a count of the number of pixels in the run, and the second value contains the value of the pixel. Runs of up to 255 identical pixel values may be encoded as only two bytes of data. Actually, it's a bit more complex than this. In addition to encoded runs, there are unencoded runs, delta markers, end-of-scan-line markers, and an end-of-RLE-data marker.

The 8-bit RLE algorithm (RLE8) stores repeating pixel values as encoded runs. The first byte of an encoded run will be in the range of 1 to 255. The second byte is the value of the 8-bit pixels in the run. For example, an encoded run of 05 18 would decode into five pixels each with the value 18, or 18 18 18 18 18.

When a scan line does not contain enough pixel runs to achieve a significant amount of compression, contiguous pixel values may be stored as literal or unencoded runs. An unencoded run may contain from 3 to 255 pixel values. The first byte of an unencoded run is always zero. This makes it possible to tell the difference between the start of an encoded and the start of an unencoded run. The second byte value is the number of unencoded pixel values that follow. If the number of pixels is odd, then a 00 padding value also follows. This padding value is not part of the original pixel data and should not be written to the decoded data stream. Here are some examples of encoded and unencoded data streams:

Ex.1007 / Page 614 of 1158
APPLE INC.

Encoded Bytes	Decoded Bytes
05 10	10 10 10 10 10
00 05 23 65 34 56 45 00	23 65 34 56 45
0A 0A	0A 0A 0A 0A 0A 0A 0A 0A 0A 0A
00 04 46 57 68 79	46 57 68 79

Three marker values may also be found in the RLE data. Each of these markers also begins with a zero-byte value. The second byte value indicates the type of marker. These markers specify positional information relating to the decoded bitmap data and do not generate any data themselves.

The first marker is the end-of-scan-line marker and is identified by two byte values 00 and 00. This marker is an indication that the end of data for the current scan line has been reached. Encoded data occurring after this marker is decoded starting at the beginning of the next scan line. If an end-of-scan-line marker is not present in the encoded data, then the pixels will automatically wrap from the end of one scan line to the start of the next.

This marker is only used when you want to force the decoding of a scan line to end at a particular place. If the end-of-scan-line marker occurs in the middle of a scan line, all remaining pixels in the decoded bitmap for the line are ignored. This "short scan line" technique is used to omit unneeded portions of scan lines. Most often, it is found in icon and pointer BMP files.

The next marker is the end of RLE data marker. It is identified by the two byte values 00 and 01. This marker occurs only as the last two bytes of the RLE data. This marker is an indication that the reader should stop decoding data.

The last marker is the run offset marker, also called a *delta* or *vector* code. This marker is four bytes in size, with the first two bytes being the values 00 and 02, and the last two values specifying a pixel address using unsigned X and Y values as an offset from the current bitmap cursor position. The X value is the number of pixels across the scan line, and the Y value is the number of rows forward in the bitmap.

This run offset marker indicates the location in the bitmap where the next decoded run of pixels should be written. For example, a run offset marker value of 00 02 05 03 would indicate that the offset of the bitmap cursor should move five pixels down the scan line, three rows forward, and write out the next run. The cursor then continues writing decoded data from its new position moving forward.

Ex.1007 / Page 615 of 1158
APPLE INC.

Run offset markers are used when a bitmap may contain a large amount of "don't care" pixels. For example, if the BMP file holds a bitmap used as a mask (such as those used with icons and pointers), many of the pixels in the rectangular bitmap may not be used. Rather than store these unused pixels in the BMP file, only the significant pixels are stored, and the delta markers are used as "jumps" to skip over the parts of the bitmap not actually used in the mask.

The following are the BMP RLE markers:

```
00 00           End of scan line
00 01           End of bitmap data
00 02 XX YY     Run offset marker
```

Here is an example of decoding an 8-bit data stream. Each of the values is an 8-bit index value into the color palette and not an actual color value.

Encoded Bytes	Decoding Description	Decoded Bytes
04 16	Four bytes of value 16	16 16 16 16
08 45	Eight bytes of value 45	45 45 45 45 45 45 45 45
00 00	End of scan line	None
00 02 04 02	Move to offset four pixels forward and two rows up	None
03 E4	Three bytes of value E4	E4 E4 E4
00 03 12 A4 46 00	Three bytes of unencoded data	12 A4 46
00 00	End of scan line	None
00 01	End of RLE data	None

The 4-bit RLE algorithm (RLE4) stores repeating pixel values in a very similar manner to RLE8. All of the markers are the same. The only real difference is that two pixel values are packed per byte, and these pixel values alternate when decoded. For example, an RLE4-encoded data stream of 07 48 would decode to seven pixels, alternating in value as 04 08 04 08 04 08 04.

If this looks bizarre, it's because you rarely see alternating runs of pixel values in bitmaps eight bits or greater in depth. Four-bit (16-color) bitmaps, however, usually contains a lot of dithering. Most dithering algorithms will yield relatively large runs of alternating pixels. Runs of repeating sequences of three and four pixels are also fairly common output from many dithering algorithms. But the ability to efficiently encode these types of pixel runs is not currently supported in the BMP RLE scheme.

In case you are thinking that runs of identical pixel values cannot be encoded by RLE4, you are incorrect. For example, a run of twelve pixels all of the value

Ex.1007 / Page 616 of 1158
APPLE INC.

9 would be RLE4-encoded as 0C 99 and would decode to the run 09 09 09 09 09 09 09 09 09 09 09 09.

Padding is added to unencoded pixel runs that are an odd number of bytes, rather than pixels, in length. And an unused final nibble in odd-length runs is set to zero. For example, the six pixel values 1 3 5 7 9 0 would be stored as the unencoded run 00 06 13 57 90 00, while the five pixel values 1 3 5 7 9 would be stored as the unencoded run 00 05 13 57 90 00.

Following is an example of decoding a 4-bit data stream. Each of the values is a 4-bit index value into the color palette and not an actual color value.

Encoded Bytes	Decoding Description	Decoded Bytes
04 16	Four values alternating 1 and 6	1 6 1 6
08 44	Eight values alternating 4 and 4	4 4 4 4 4 4 4 4
00 00	End of scan line	None
00 02 04 02	Move to offset four pixels forward and two rows up	None
03 E4	Three values alternating E and 4	E 4 E
00 06 12 A4 46 00	Six values of unencoded data	1 2 A 4 4 6
00 00	End of scan line	None
00 01	End of RLE data	None

Here is a summary of Windows BMP data characteristics:

Pixel Depth	Pixel Size	Compression	Color Palette	Color Masks
1 bit	1 bit	0	Yes	No
4 bits	4 bits	0,2	Yes	No
8 bits	1 byte	0,1	Yes	No
16 bits	4 bytes	3	No	Yes
24 bits	3 bytes	0	No	No
32 bits	4 bytes	3	No	Yes

For Further Information

For further information about the Microsoft Windows Bitmap format, see the documentation included on the CD-ROM that accompanies this book. Although you probably will not be able to get any information directly from them, here is Microsoft's address:

Ex.1007 / Page 617 of 1158
APPLE INC.

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Voice: 206-882-8080
FAX: 206-936-7329
BBS: 206-637-9009
FTP: *ftp://ftp.microsoft.com*
WWW: *http://www.microsoft.com*

The closest thing there is to an archive of Microsoft file format specifications is the Microsoft Developers Network Knowledge Base available on the MSDN CD-ROM and at Microsoft's Web site. The Win16 and Win32 Software Development Kits (SDKs) also have information on BMP.

Information about the Windows BMP format can also be found in the following references:

Charlap, David, "The BMP File Format: Part I," *Dr. Dobb's Journal*, vol. 20, no. 228, March 1995.

Charlap, David, "The BMP File Format: Part II," *Dr. Dobb's Journal*, vol. 20, no. 229, April 1995.

Luse, Marv, "The BMP File Format," *Dr. Dobb's Journal*, vol. 9, no. 219, September 1994, pp. 18–22.

Microsoft Corporation, *Microsoft Win32 Programmer's Reference*, vol. 5, Microsoft Press, Redmond, WA, 1993.

Microsoft Corporation, *Microsoft Windows Programmer's Reference*, vol. 2, v3, Microsoft Press, Redmond, WA, 1990.

Petzold, Charles, "Preserving a Device-Independent Bitmap: The Packed-DIB Format," *PC Magazine*, July 1991, pp. 433–439.

Petzold, Charles, "What's New in Bitmap Formats: A Look at Windows and OS/2," *PC Magazine*, 11 September 1990, pp. 403–410.

Swan, Tom, *Inside Windows File Formats*, Sams Publishing, 1993.

The code for the above issues of *Dr. Dobb's Journal* are available at:

ftp://ftp.mv.com/pub/ddj/1994/1194.09/bmp.zip
ftp://ftp.mv.com/pub/ddj/1995/1195.03/bmp.zip

Ex.1007 / Page 618 of 1158
APPLE INC.

The two *Dr. Dobb's Journal* articles by David Charlap contain a complete collection of source code for working with Windows 2.*x*, 3.*x*, NT, and OS/2 BMP file formats. It is available at the above FTP site and on this book's CD-ROM.

Ex.1007 / Page 619 of 1158
APPLE INC.

Microsoft Windows Metafile

NAME:	Microsoft Windows Metafile
ALSO KNOWN AS:	Windows Metafile, WMF
TYPE:	Metafile
COLORS:	24-bit maximum
COMPRESSION:	NA
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	Little-endian
ORIGINATOR:	Microsoft Corporation
PLATFORM:	Microsoft Windows
SUPPORTING APPLICATIONS:	Numerous Microsoft Windows-based graphics applications
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	Encapsulated PostScript, Microsoft Windows Bitmap

USAGE: Used for file interchange, device support.

COMMENTS: A widely used format associated with Microsoft Windows, although applications on other platforms may provide support.

Overview

Microsoft Windows Metafile (WMF) files are used to store vector and bitmap-format image data in memory or in disk files for later playback to an output device. Although Windows Metafile is specific to Microsoft Windows, many non-Windows-based applications support this format as a method for interchanging data with Windows applications. Because of the widespread popularity of the Microsoft Windows GUI, the Windows Metafile format has become a staple format for graphical applications and is supported on all platforms. Encapsulated PostScript (EPSF) supports the use of an included Windows Metafile when required to store vector-based data. The logical unit of measurement used in Windows Metafiles is the *twip*. A twip (meaning "twentieth of a point") is equal to 1/1440 of an inch. Thus 720 twips equal 1/2 inch, while 32,768 twips is 22.75 inches.

Ex.1007 / Page 620 of 1158
APPLE INC.

File Organization

Windows Metafile format files contain a header, followed by one or more records of data. The header contains a description of the record data stored in the metafile. Each record is a binary-encoded Microsoft Windows Graphics Device Interface (GDI) function call. The GDI is used by Windows to perform all output to a screen window or other output device. When the metafile data is rendered (or played back, in Microsoft terminology), the data from each record is used to perform the appropriate function call to render each object in the image. The last record in the file contains information indicating that the end of the record data has been reached.

File Details

The header is 18 bytes in length and is structured as follows:

```
typedef struct _WindowsMetaHeader
{
  WORD  FileType;       /* Type of metafile (1=memory, 2=disk) */
  WORD  HeaderSize;     /* Size of header in WORDS (always 9). */
  WORD  Version;        /* Version of Microsoft Windows used */
  DWORD FileSize;       /* Total size of the metafi+le in WORDs */
  WORD  NumOfObjects;   /* Number of objects in the file */
  DWORD MaxRecordSize;  /* The size of largest record in WORDs */
  WORD  NoParameters;   /* Not Used (always 0) */

} WMFHEAD;
```

FileType contains a value which indicates the location of the metafile data. A value of 1 indicates that the metafile is stored in memory, while a 2 indicates that it is stored on disk.

HeaderSize contains the size of the metafile header in WORDs.

Version stores the version number of Microsoft Windows that created the metafile. This value is always read in hexadecimal format. For example, in a metafile created by Windows 3.0, this item would have the value 300h.

FileSize specifies the total size of the metafile in 16-bit WORDs.

NumOfObjects specifies the number of objects that are in the metafile.

MaxRecordSize specifies the size of the largest record in the metafile in WORDs.

NumOfParams is not used and is set to a value of 0.

Ex.1007 / Page 621 of 1158
APPLE INC.

Following the header is a series of data records. The basic format of each record is shown below:

```
typedef struct _WindowsMetaRecord
{
DWORD Size;          /* Total size of the record in WORDs */
WORD  Function;      /* Function number (defined in WINDOWS.H) */
WORD  Parmeters[];   /* Parameter values passed to function */

} WMFRECORD;
```

Size is the total size of the records in WORDs, including the Size field itself. The minimum possible size for a record is 3.

Function is the GDI number of the function.

Parameters is an array of the parameters used by the function. The parameters are stored in the reverse order in which they are passed to the function.

When a Windows Metafile format file is played back, each record is read and the function call it contains is executed in the sequence in which it is read. The last record in every metafile always has a function number of zero and is used to indicate the end of the record data.

There are several important considerations that must be observed when reading WMF record data.

First, not all of the records in a Windows Metafile have the above format, although most do. The GDI function calls that do follow the basic record format are the following:

Arc	RealizePalette	SetPolyFillMode
Chord	Rectangle	SetROP2
Ellipse	ResizePalette	SetStretchBltMode
ExcludeClipRect	RestoreDC	SetTextAlign
FloodFill	RoundRect	SetTextCharExtra
IntersectClipRect	SaveDC	SetTextColor
LineTo	ScaleViewportExt	SetTextJustification
MoveTo	ScaleWindowExt	SetViewportExt
OffsetClipRgn	SetBkColor	SetViewportOrg
OffsetViewportOrg	SetBkMode	SetWindowExt
OffsetWindowOrg	SetMapMode	SetWindowOrg
PatBlt	SetMapperFlags	
Pie	SetPixel	

Ex.1007 / Page 622 of 1158
APPLE INC.

Second, several record formats deviate from this basic record format by containing a data structure, rather than a data array, in the Parameters field. These are:

AnimatePalette	CreatePatternBrush	Escape
BitBlt	CreatePenIndirect	ExtTextOut
CreateBrushIndirect	CreateRegion	Polygon
CreateFontIndirect	DeleteObject	PolyPolygon
CreatePalette	DrawText	Polyline

Consult the Microsoft Windows Programmer's Reference Library for the internal structure of each of these special records.

Third, several GDI function calls were added or had their parameters changed with the release of Microsoft Windows 3.0. GDI function calls in this category include:

AnimatePalette	Record	DeleteObject
BitBlt	CreatePatternBrush	RealizePalette
CreatePalette	Record	ResizePalette

Note that not all GDI function calls can appear in a metafile. The only calls that are valid are those that take a handle to a device context as their first parameter. A complete list of all of the GDI function calls is documented in *Microsoft Windows Programmer's Reference.* They are also found in the *WINDOWS.H* header file. These GDI function calls are the directives that begin with the characters META. There are more than 70 different GDI function calls defined for Windows 3.0.

Porting WMF Files Between Applications

Most Microsoft Windows applications that create metafiles prepend a 22-byte header to the file. This header contains information not found in the metafile header, but which is needed to move the metafile information between applications. The structure of this header is as follows:

```
typedef struct _WmfSpecialHeader
{
  DWORD Key;            /* Magic number (always 9AC6CDD7h) */
  WORD  Handle;         /* Metafile HANDLE number (always 0) */
  SHORT Left;           /* Left coordinate in metafile units */
  SHORT Top;            /* Top coordinate in metafile units */
  SHORT Right;          /* Right coordinate in metafile units */
  SHORT Bottom;         /* Bottom coordinate in metafile units */
```


Ex.1007 / Page 623 of 1158
APPLE INC.

```
    WORD  Inch;          /* Number of metafile units per inch */
    DWORD Reserved;      /* Reserved (always 0) */
    WORD  Checksum;      /* Checksum value for previous 10 WORDs */

} WMFSPECIAL;
```

Key contains a special identification value that indicates the presence of a special header and is always 9AC6CDD7h.

Handle is not used and always contains the value 0.

Left, Top, Right, and Bottom contain the coordinates of the upper-left and lower-right corners of the image on the output device. These are measured in twips. These four fields also correspond to the RECT structure used in Microsoft Windows and found in the file *WINDOWS.H*.

Inch contains the number of twips per inch used to represent the image. Normally, there are 1440 twips per inch; however, this number may be changed to scale the image. A value of 720 indicates that the image is double its normal size, or scaled to a factor of 2:1. A value of 360 indicates a scale of 4:1, while a value of 2880 indicates that the image is scaled down in size by a factor of two. A value of 1440 indicates a 1:1 scale ratio.

Reserved is not used and is always set to 0.

Checksum contains a checksum value for the previous 10 WORDs in the header, calculated by XORing each WORD value to 0:

```
WMFSPECIAL wmfspecial; wmfspecial.
Checksum = 0;

wmfspecial.Checksum ^= (wmfspecial.Key & 0x0000FFFFL);
wmfspecial.Checksum ^= ((wmfspecial.Key & 0xFFFF0000L) >> 16);
wmfspecial.Checksum ^= wmfspecial.Handle; wmfspecial.Checksum ^=
wmfspecial.Left;
wmfspecial.Checksum ^= wmfspecial.Top; wmfspecial.Checksum ^=
wmfspecial.Right;
wmfspecial.Checksum ^= wmfspecial.Bottom; wmfspecial.Checksum ^=
wmfspecial.Inch;
wmfspecial.Checksum ^= (wmfspecial.Reserved & 0x0000FFFFL);
wmfspecial.Checksum ^= ((wmfspecial.Reserved & 0xFFFF0000L) >> 16);
```

An alternative way to step through the header structure one WORD at a time is to use a pointer as shown below:

```
WMFSPECIAL *wmfspecial;
WORD *ptr;
wmfspecial->Checksum = 0;
for(ptr = (WORD *) wmfspecial;
    ptr < (WORD *)wmfspecial->Checksum;
```


Ex.1007 / Page 624 of 1158
APPLE INC.

```
    ptr++)
        wmfspecial->Checksum ^= *ptr;
```

Storing Bitmaps in a WMF File

The BitBlt function (GDI function number 940h) is used to store device-independent bitmaps in a Windows Metafile. This record was modified for Windows 3.0, so metafiles created under earlier versions of Windows may not be suitable for playback on all graphics output devices.

For Further Information

For further information about the Microsoft Windows Metafile format, see the specification included on the CD-ROM that accompanies this book. You may also obtain information by contacting Microsoft at:

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Voice: 206-882-8080
FAX: 206-936-7329
BBS: 206-637-9009
WWW: *http://www.microsoft.com/*

Additional information about the Windows Metafile Format and the Microsoft Windows Graphics Device Interface can also be found in the following references:

Petzold, Charles, *Programming Windows: The Microsoft Guide to Writing Applications for Windows 3*, Second Edition, Microsoft Press, Redmond, WA, 1990.

Microsoft Corporation, *Microsoft Windows: A Guide to Programming*, Microsoft Windows Programmer's Reference Library, Microsoft Press, Redmond, WA, 1990.

Microsoft Corporation, *Microsoft Windows: Programmer's Reference*, Microsoft Windows Programmer's Reference Library, Microsoft Press, Redmond, WA, 1990.

Microsoft Corporation, *Microsoft Windows: Programming Tools*, Microsoft Windows Programmer's Reference Library, Microsoft Press, Redmond, WA, 1990.

Ex.1007 / Page 625 of 1158
APPLE INC.

The Microsoft Windows Programmer's Reference Library is the master reference for programmers working with all aspects of Microsoft Windows. The books in this library are supplied with the Microsoft Windows Software Development Kit (SDK). The manuals supplied with the Microsoft C 7.0 Professional Development Systems are also very helpful. You can get information about obtaining these products from:

Microsoft Information Center
Voice: 800-426-9400

You may also be able to get information via FTP through the Developer Relations Group at:

ftp://ftp.microsoft.com/developer/drg/

Ex.1007 / Page 626 of 1158
APPLE INC.

NAME:	MIFF
ALSO KNOWN AS:	Machine Independent File Format
TYPE:	Bitmap
COLORS:	16 million
COMPRESSION:	RLE, Q-coder, JPEG predictive arithmetic compression
MAXIMUM IMAGE SIZE:	Unlimited
MULTIPLE IMAGES PER FILE:	No
NUMERICAL FORMAT:	NA
ORIGINATOR:	John Cristy
PLATFORM:	X Window System
SUPPORTING APPLICATIONS:	ImageMagick
SPECIFICATION ON CD:	Yes
CODE ON CD:	Yes (in ImageMagick package)
IMAGES ON CD:	Yes
SEE ALSO:	JFIF

USAGE: Bitmap still image and animation storage format.

COMMENTS: MIFF is the native image file format for the X Window System-based ImageMagick utilities.

Overview

MIFF (Machine Independent File Format) is a platform-independent format for storing bitmap images. MIFF is part of the ImageMagick toolkit of image manipulation utilities for the X Window System. ImageMagick is capable of converting many different image file formats to and from MIFF, in addition to creating and displaying animated bitmap image presentations.

File Organization

The MIFF header is composed entirely of ASCII characters. The fields in the header are keyword and value combinations in the *keyword=value* format, with each keyword and value separated by an equal sign (=).

Ex.1007 / Page 627 of 1158
APPLE INC.

Each *keyword=value* combination is delimited by at least one control or white-space character. Comments may appear in the header section and are always delimited by braces. The MIFF header always ends with a colon (:) character, followed by a newline character. It is also common for a formfeed and a newline character to appear before the colon.

The following is a list of *keyword=value* combinations that may be found in a MIFF file:

class=DirectClass or class=PseudoClass
: class indicates the type of binary image data stored in the MIFF file. If this keyword is not present, DirectClass image data is assumed.

colors=*value*
: colors specifies the number of colors in a DirectClass image. For a PseudoColor image this keyword specifies the size of the colormap. If this keyword is not present in the header, and the image is PseudoColor, then a linear colormap is used with the image data.

columns=*value*
: columns indicates the width of the image in pixels. This is a required keyword and has no default.

compression=QEncoded or compression=RunlengthEncoded
: compression indicates the type of algorithm used to compress the image data. If this keyword is not present, the image data is assumed to be uncompressed.

id=ImageMagick
: The id keyword identifies the file as a MIFF-format image file. This keyword is required and has no default.

packets=*value*
: packets specifies the number of compressed color packets in the image data section. This keyword is optional for RunlengthEncoded images, mandatory for QEncoded images, and not used for uncompressed images.

rows=*value*
: rows indicates the height of the image in pixels. This is a required keyword and has no default.

Ex.1007 / Page 628 of 1158
APPLE INC.

scene=*value*

scene indicates the sequence number for this MIFF image file. This optional keyword is used when a MIFF image file is one in a sequence of files used in an animation.

signature=*value*

The optional keyword signature contains a string that uniquely identifies the image colormap. Unique colormap identifiers are normally used when animating a sequence of PseudoClass images.

The following is a sample MIFF header. In this example, <FF> is a formfeed character:

```
{
    A sample MIFF header
}
id=ImageMagick
class=PseudoClass  colors=256
compression=RunlengthEncoded  packets=10672
columns=800  rows=600 {size of the image}
scene=1  signature=d79e1c308aa5bbcdeea8ed63df412da9
<FF>
:
```

Note that *keyword=value* combinations may be separated by newlines or spaces and may occur in any order within the header. Comments (within braces) may appear anywhere before the colon.

Following the header is the binary image data itself. How the image data is formatted depends upon the class of the image as specified (or not specified) by the value of the class keyword in the header.

File Details

DirectClass images (class=DirectClass) are continuous tone, RGB images stored as intensity values in red-green-blue order. Each color value is one byte in size and there are three bytes per pixel. The total number of pixels in a DirectClass image is calculated by multiplying the rows value by the columns value in the header.

PseudoClass images (class=PseudoClass) are colormapped RGB images. The colormap is stored as a series of red-green-blue pixel values, each value being a byte in size. The number of map entries is indicated by the colors keyword in the header, with a maximum of 65,535 total entries allowed. The colormap data occurs immediately following the header.

Ex.1007 / Page 629 of 1158
APPLE INC.

PseudoClass image data is an array of index values into the color map. If there are 256 or fewer colors in the image, each byte of image data contains an index value. If the image contains more than 256 colors, then the index value is stored in two contiguous bytes with the most significant byte being first. The total number of pixels in a PseudoClass image is calculated by multiplying the rows value by the columns value in the header.

MIFF is capable of storing a digital signature for colormapped images. This signature was developed for use when animating a sequence of images on a colormapped X server. All of the signatures in a sequence of MIFF files are checked, and if they all match, you do not need to compute a global colormap.

The default colormap identifier is a digital signature computed using the RSA Data Security MD4 Digest Algorithm. (See a description of this algorithm in RFC 1186, October 1990.) The colormap signature is computed if the MIFF file is part of a scene (i.e., the scene value does not equal 0).

The image data in a MIFF file may be uncompressed or may be compressed using one of two algorithms. The predictive arithmetic compression algorithm found in the JPEG compression scheme (Chapter 9, *Data Compression*) may be used to encode either DirectColor or PseudoColor image data into packets of compressed data. Older MIFF files will use the IBM Q-coder algorithm to QEncode image data. The number of Q-encoded packets stored in the file is specified by the packets keyword in the header.

A less costly alternative to the Q-coder algorithms is a simple, run-length encoding (RLE) algorithm. For DirectColor images, runs of identical pixel values (not BYTE values) are encoded into a series of four-byte packets. The first three bytes of the packet contain the red, green, and blue values of the pixel in the run. The fourth byte contains the number of pixels in the run. This value may be in the range of 0 to 255 and is one less than the actual number of pixels in the run. For example, a value of 147 indicates that there are 148 pixels in the run.

For PseudoColor images, the same RLE algorithm is used. Runs of identical index values are encoded into packets. Each packet contains the colormap index value followed by the number of index values in the run. The number of bytes in a PseudoColor RLE packet will be either two or three, depending upon the size of the index values. The number of RLE packets stored in the file is specified by the packets keyword in the header, but is not required.

Ex.1007 / Page 630 of 1158
APPLE INC.

For Further Information

For further information about MIFF, see the specification included on the CD-ROM that accompanies this book.

ImageMagick was created by John Cristy, of E.I. duPont de Nemours & Company, and is copyright by duPont. The software is included on the CD-ROM that accompanies this book. See the following sites for ImageMagick:

http://www.wizards.dupont.com/cristy/ImageMagick.html
ImageMagick homepage

http://web.cnam.fr/Images/Usenet/docs/imagemagick.html
What is ImageMagick?

ftp://ftp.wizards.dupont.com/pub/ImageMagick/
ImageMagick FTP site

For more information about MIFF, you can contact:

duPont de Nemour & Company
Attn: John Cristy
Central Research and Development
Experimental Station
P.O. Box 80328
Room 162-A
Wilmington, DE 19880-0328
Voice: 302-695-1159
Email: *cristy@dupont.com*

For information about the JPEG predictive arithmetic compression algorithm, see the section called "JPEG Compression" in Chapter 9.

For information about the IBM-patented Q-coder compression algorithm, see the following reference:

Mitchell, J. L. and W.B. Pennebaker, "Software Implementations of the Q-Coder," *IBM Journal of Research Development*, vol. 32, no. 6, November 1988, pp. 753–74.

Ex.1007 / Page 631 of 1158
APPLE INC.

MPEG

NAME:	MPEG
ALSO KNOWN AS:	MPG, MPEG-1, MPEG-2
TYPE:	Audio/video data storage
COLORS:	Up to 24-bits (4:2:0 YCbCr color space)
COMPRESSION:	DCT and block-based scheme with motion compensation
MAXIMUM IMAGE SIZE:	4095×4095×30 frames/second
MULTIPLE IMAGES PER FILE:	Yes (multiple program multiplexing)
NUMERICAL FORMAT:	NA
ORIGINATOR:	Motion Picture Experts Group (MPEG) of the International Standards Organization (ISO)
PLATFORM:	All
SUPPORTING APPLICATIONS:	Xing Technologies MPEG player, others
SPECIFICATION ON CD:	Yes (FAQ)
CODE ON CD:	Yes (in ISO MPEG-2 Codec package)
IMAGES ON CD:	Yes
SEE ALSO:	JPEG File Interchange Format, Intel DVI

USAGE: Stores an MPEG-encoded data stream on a digital storage medium. MPEG is used to encode audio, video, text, and graphical data within a single, synchronized data stream.

COMMENTS: MPEG-1 is a finalized standard in wide use. MPEG-2 is still in the development phase and continues to be revised for a wider base of applications. Currently, there are few stable products available for making practical use of the MPEG standard, but this is changing.

Overview

MPEG (pronounced "em-peg") is an acronym for the Motion Picture Experts Group, a working group of the International Standards Organization (ISO) that is responsible for creating standards for digital video and audio compression.

Ex.1007 / Page 632 of 1158
APPLE INC.

The MPEG specification is a specification for an encoded data stream which contains compressed audio and video information. MPEG was designed specifically to store sound and motion-video data on standard audio Compact Discs (CD) and Digital Audio Tapes (DAT).

The main application for MPEG is the storage of audio and video data on CD-ROMs for use in multimedia systems, such as those found on the Apple Macintosh platform and in the Microsoft Windows environment. Such systems require the ability to store and play back high-quality audio and video material for commercial, educational, and recreational applications. The new MPEG-2 standard allows the transmission of MPEG data across television and cable network systems.

On most systems, you use special hardware to capture MPEG data from a live video source at a real-time sampling rate of 30 frames per second. Each frame of captured video data is then compressed and stored as an MPEG data stream. If an audio source is also being sampled, it too is encoded and multiplexed in with the video stream, with some extra information to synchronize the two streams together for playback.

To play back MPEG data, you use either a hardware/software or software-only player. The player reads in the MPEG data stream, decompresses the information, and sends it to the display and audio systems of the computer. Speed of the playback depends upon how quickly the resources of the computer allow the MPEG data to be read, decompressed, and played. Available memory, CPU speed, and disk I/O throughput are all contributing factors. The quality of the MPEG stream is determined during encoding, and there are typically no adjustments available to allow an application to "tweak" the apparent quality of the MPEG output produced during playback.

MPEG is based on digital television standards (specified in CCIR-601) used in the United States. In its initial form, MPEG is not actually capable of storing CCIR-601 images. The typical resolution of 720×576 requires more bandwidth than the maximum MPEG data rate of 1.86Mbits/second allows. Standard television images must therefore be decimated by 2:1 into lower resolution SIF format data (352×240) to be stored.

European (PAL and SECAM) and Japanese standards are different in many respects, including the display rate (30 frames/second U.S., 25 frames/second European) and the number of lines per field (240 U.S., 288 European). Therefore, an MPEG player must be able to recognize a wide variety of variations possible in the encoded video signal itself.

Ex.1007 / Page 633 of 1158
APPLE INC.

Constrained Parameters Bitstreams (CPB) are a complex aspect of MPEG. CPBs are those bitstreams that are limited in terms of picture size, frame rate, and coded bit-rate parameters. These limitations normalize the computation complexity required of both hardware and software, thus guaranteeing a reasonable, nominal subset of MPEG that can be decoded by the widest possible range of applications while still remaining cost-effective. MPEG bitstreams for video are limited to 1.86 Mbits/second if they meet constrained parameters. If it were not for the constrained parameters, the MPEG syntax could specify a data rate of more than 100 Mbits/second.

File Organization

No actual structured MPEG file format has been defined. Everything required to play back MPEG data is encoded directly in the data stream. Therefore, no header or other type of wrapper is necessary. It is likely that when needed, a multimedia standards committee—perhaps MHEG or the DSM (Digital Storage Medium) MPEG subgroup—will one day define an MPEG file format.

File Details

This section describes the relationship between MPEG, JPEG, and MJPEG, the type of compression used for MPEG files, and the MPEG-2 standard.

Relationship Between MPEG, JPEG, and MJPEG

Some people are confused about the relationship between MPEG and JPEG. The MPEG and JPEG (Joint Photographic Experts Group) committees of the ISO originally started as the same group, but with two different purposes. JPEG focused exclusively on still-image compression, while MPEG focused on the encoding/synchronization of audio and video signals within a single data stream. Although MPEG employs a method of spatial data compression similar to that used for JPEG, they are not the same standard nor were they designed for the same purpose.

Another acronym you may hear is MJPEG (Motion JPEG). Several companies have come out with an alternative to MPEG—a simpler solution (but not yet a standard) for how to store motion video. This solution, called Motion JPEG, simply uses a digital video capture device to sample a video signal, to capture frames, and to compress each frame in its entirety using the JPEG compression method. A Motion JPEG data stream is then played back by decompressing and

Ex.1007 / Page 634 of 1158
APPLE INC.

displaying each individual frame. A standard audio compression method is usually included in the Motion JPEG data stream.

There are several advantages to using Motion JPEG:

- Fast, real-time compression rate
- No frame-to-frame interpolation (motion compensation) of data is required

But there are also disadvantages:

- Motion JPEG files are considerably larger than MPEG files
- They are somewhat slower to play back (more information per frame than MPEG)
- They exhibit poor video quality if a higher JPEG compression ratio (quality factor) is used

On average, the temporal compression method used by MPEG provides a compression ratio three times that of JPEG for the same perceived picture quality.

MPEG Compression

MPEG uses an asymmetric compression method. Compression under MPEG is far more complicated than decompression, making MPEG a good choice for applications that need to write data only once, but need to read it many times. An example of such an application is an archiving system. Systems that require audio and video data to be written many times, such as an editing system, are not good choices for MPEG; they will run more slowly when using the MPEG compression scheme.

MPEG uses two types of compression methods to encode video data: interframe and intraframe encoding. Interframe encoding is based upon both predictive coding and interpolative coding techniques, as described below.

When capturing frames at a rapid rate (typically 30 frames/second for real time video) there will be a lot of identical data contained in any two or more adjacent frames. If a motion compression method is aware of this "temporal redundancy," as many audio and video compression methods are, then it need not encode the entire frame of data, as is done via *intraframe encoding*. Instead, only the differences (deltas) in information between the frames is encoded. This results in greater compression ratios, with far less data needing to be encoded. This type of *interframe encoding* is called *predictive encoding*.

Ex.1007 / Page 635 of 1158
APPLE INC.

A further reduction in data size may be achieved by the use of bi-directional prediction. Differential predictive encoding encodes only the differences between the current frame and the previous frame. Bi-directional prediction encodes the current frame based on the differences between the current, previous, and next frame of the video data. This type of interframe encoding is called motion-compensated interpolative encoding.

To support both interframe and intraframe encoding, an MPEG data stream contains three types of coded frames:

- I-frames (intraframe encoded)
- P-frames (predictive encoded)
- B-frames (bi-directional encoded)

An *I-frame* contains a single frame of video data that does not rely on the information in any other frame to be encoded or decoded. Each MPEG data stream starts with an I-frame.

A *P-frame* is constructed by predicting the difference between the current frame and closest preceding I- or P-frame. A *B-frame* is constructed from the two closest I- or P-frames. The B-frame must be positioned between these I- or P-frames.

A typical sequence of frames in an MPEG stream might look like this:

```
IBBPBBPBBPBBIBBPBBPBBPBBI
```

In theory, the number of B-frames that may occur between any two I- and P-frames is unlimited. In practice, however, there are typically twelve P- and B-frames occurring between each I-frame. One I-frame will occur approximately every 0.4 seconds of video runtime.

Remember that the MPEG data is not decoded and displayed in the order that the frames appear within the stream. Because B-frames rely on two reference frames for prediction, both reference frames need to be decoded first from the bitstream, even though the display order may have a B-frame in between the two reference frames.

In the previous example, the I-frame is decoded first. But, before the two B-frames can be decoded, the P-frame must be decoded, and stored in memory with the I-frame. Only then may the two B-frames be decoded from the information found in the decoded I- and P-frames. Assume, in this example, that

Ex.1007 / Page 636 of 1158
APPLE INC.

you are at the start of the MPEG data stream. The first ten frames are stored in the sequence IBBPBBPBBP (0123456789), but are decoded in the sequence:

```
IPBBPBBPBB (0312645978)
```

and finally are displayed in the sequence:

```
IBBPBBPBBP (0123456789)
```

Once an I-, P-, or B-frame is constructed, it is compressed using a DCT compression method similar to JPEG. Where interframe encoding reduces temporal redundancy (data identical over time), the DCT-encoding reduces spatial redundancy (data correlated within a given space). Both the temporal and the spatial encoding information are stored within the MPEG data stream.

By combining spatial and temporal subsampling, the overall bandwidth reduction achieved by MPEG can be considered to be upwards of 200:1. However, with respect to the final input source format, the useful compression ratio tends to be between 16:1 and 40:1. The ratio depends upon what the encoding application deems as "acceptable" image quality (higher quality video results in poorer compression ratios). Beyond these figures, the MPEG method becomes inappropriate for an application.

In practice, the sizes of the frames tend to be 150 Kbits for I-frames, around 50 Kbits for P-frames, and 20 Kbits for B-frames. The video data rate is typically constrained to 1.15 Mbits/second, the standard for DATs and CD-ROMs.

The MPEG standard does not mandate the use of P- and B-frames. Many MPEG encoders avoid the extra overhead of B- and P-frames by encoding I-frames. Each video frame is captured, compressed, and stored in its entirety, in a similar way to Motion JPEG. I-frames are very similar to JPEG-encoded frames. In fact, the JPEG Committee has plans to add MPEG I-frame methods to an enhanced version of JPEG, possibly to be known as JPEG-II.

With no delta comparisons to be made, encoding may be performed quickly; with a little hardware assistance, encoding can occur in real time (30 frames/second). Also, random access of the encoded data stream is very fast because I-frames are not as complex and time-consuming to decode as P- and B-frames. Any reference frame needs to be decoded before it can be used as a reference by another frame.

There are also some disadvantages to this scheme. The compression ratio of an I-frame-only MPEG file will be lower than the same MPEG file using motion

Ex.1007 / Page 637 of 1158
APPLE INC.

compensation. A one-minute file consisting of 1800 frames would be approximately 2.5Mb in size. The same file encoded using B- and P-frames would be considerably smaller, depending upon the content of the video data. Also, this scheme of MPEG encoding might decompress more slowly on applications that allocate an insufficient amount of buffer space to handle a constant stream of I-frame data.

MPEG-2

The original MPEG standard is now referred to as MPEG-1. The MPEG-1 Video Standard is aimed at small-scale systems using CD-ROM storage and small, lower resolution displays. Its 1.5-Megabit/second data rate, however, limits MPEG-1 from many high-power applications. The next phase in MPEG technology development is MPEG-2.

The new MPEG-2 standard is a form of digital audio and video designed for the television industry. It will be used primarily as a way to consolidate and unify the needs of cable, satellite, and television broadcasts, as well as computing, optical storage, Ethernet, VCR, CD-I, HDTV, and blue-laser CD-ROM systems.

MPEG-2 is an extension of the MPEG-1 specification and therefore shares many of the same design features. The baseline part of MPEG-2 is called the Video Main Profile and provides a minimum definition of data quality. This definition fills the needs of high-quality television program distribution over a wide variety of data networks. Video Main Profile service over cable and satellite systems could possibly start in 1994. Consumers who need such features as interactive television and vision phones will benefit greatly from this service.

Features added by MPEG-2 include:

- Interlaced video formats
- Multiple picture aspect ratios (such as 4:3 and 16:9, as required by HDTV)
- Conservation of memory usage (by lowering the picture quality below the Video Main Profile definition)
- Increased video quality over MPEG-1 (when coding for the same target arbitrates)
- Ability to decode MPEG-1 data streams.

MPEG-2 can also multiplex audio, video, and other information into a single data stream and provides 2- to 15-Mbits/second data rates while maintaining full CCIR-601 image quality. MPEG-2 achieves this by the use of two types of data streams: the Program stream and the Transport stream.

Ex.1007 / Page 638 of 1158
APPLE INC.

The Program stream is similar to the MPEG-1 System stream, with extensions for encoding program-specific information, such as multiple language audio channels. The Transport stream was newly added to MPEG-2 and is used in broadcasting by multiplexing multiple programs comprised of audio, video, and private data, such as combining standard-definition TV and HDTV signals on the same channel. MPEG-2 supports multi-program broadcasts, storage of programs on VCRs, error detection and correction, and synchronization of data streams over complex networks.

Just as MPEG-1 encoding and decoding hardware has appeared, so will the same hardware for MPEG-2. With its broad range of applications and its toolkit approach, MPEG-2 encoding and decoding is very difficult to implement fully in a single chip. A "do everything" MPEG-2 chipset is not only difficult to design, but also expensive to sell. It is more likely that MPEG-2 hardware designed for specific applications will appear in the near future, with much more extensible chipsets to come in the more distant future.

The compression used on the MPEG audio stream data is based on the European MUSICAM standard, with additional pieces taken from other algorithms. It is similar in conception to the method used to compress MPEG video data. It is a lossy compression scheme, which throws away (or at least assigns fewer bits of resolution to) audio data that humans cannot hear. It is also a temporal-based compression method, compressing the differences between audio samples rather than the samples themselves. At this writing, a publicly available version of the audio code was due to be released by the MPEG audio group.

The typical bandwidth of a CD audio stream is 1.5 Mbits/second. MPEG audio compression can reduce this data down to approximately 256 Kbits/second for a 6:1 compression ratio with no discernible loss in quality (lower reductions are also possible). The remaining 1.25 Mbits/second of the bandwidth contain the MPEG-1 video and system streams. And using basically the same MPEG-1 audio algorithm, MPEG-2 audio will add discrete surround sound channels.

For Further Information

For further information about MPEG, see the MPEG Frequently Asked Questions (FAQ) document included on the CD-ROM that accompanies this book. Note, however, that this FAQ is included for background only; because it is constantly updated, you should obtain a more recent version. The MPEG FAQ on USENET is posted monthly to the newsgroups *comp.graphics, comp.compression,*

Ex.1007 / Page 639 of 1158
APPLE INC.

and *comp.multimedia.* The FAQ is available by using FTP from *rtfm.mit.edu* and is located in the directories that are called */pub/usenet/comp.graphics* and */pub/usenet/comp.compression.*

To obtain the full MPEG draft standard, you will have to purchase it from ANSI. The MPEG draft ISO standard is ISO CD 11172. This draft contains four parts:

11172.1 Synchronization and multiplexing of audio-visual information
11172.2 Video compression
11172.3 Audio compression
11172.4 Conformance testing

Contact ANSI at:

American National Standards Institute
Sales Department
1430 Broadway
New York, NY, 10018
Voice: 212-642-4900

Drafts of the MPEG-2 standard are expected to be available soon. For more information about MPEG, see the following article:

Le Gall, Didier, "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM,* vol. 3, no. 4, April 1991, pp. 46–58.

On the CD-ROM you will find several pieces of MPEG software. The ISO MPEG-2 Codec software, which converts uncompressed video frames into MPEG-1 and MPEG-2 video-coded bitstream sequences, and vice versa, is included in source code form and as a precompiled MS-DOS binary. The Sparkle MPEG player is also included for Macintosh platforms.

Ex.1007 / Page 640 of 1158
APPLE INC.

NAME:	MTV
ALSO KNOWN AS:	None
TYPE:	Scene description
COLORS:	NA
COMPRESSION:	Uncompressed
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	ASCII
ORIGINATOR:	Mark VandeWettering
PLATFORM:	All
SUPPORTING APPLICATIONS:	MTV, various conversion utilities
SPECIFICATION ON CD:	Yes
CODE ON CD:	No
IMAGES ON CD:	No
SEE ALSO:	NFF, PBM

USAGE: Used by the MTV ray tracer, although numerous conversion utilities, usually associated with other ray tracers, exist.

COMMENTS: One of the early ray tracers, still in use because of its easy availability, wide distribution, and simplicity.

Overview

The MTV formats were created to support Mark VandeWettering's MTV ray tracer and are named for the author. The MTV application has been ported to many platforms and has enjoyed wide distribution, through the *comp.graphics* newsgroup on the Internet and through the network of private (primarily PC-based) BBSs. Although the author considers both the program and the format to be dead, the format still enjoys a certain degree of currency, mainly due to its understandable design and simplicity. MTV is still being downloaded with some regularity from a number of bulletin boards and information services.

Ex.1007 / Page 641 of 1158
APPLE INC.

File Organization and Details

Both the MTV input format and the output format are based on other formats that are described in this book. The following sections provide summary information only.

Input Format

The MTV input format is identical to the Neutral File Format (NFF) developed by Eric Haines and described in detail in the NFF article.

NFF files consist of lines of ASCII text. Each line describes an object called an entity. The first field of each line describes the entity's type, and subsequent fields on the same line, and possibly subsequent lines, contain further information about the entity. The following entities are currently supported:

- Simple perspective frustum
- Background color description
- Positional (versus directional) light-source description
- Surface-properties description
- Polygon, polygonal patch, cylinder/cone, and sphere descriptions

Entities are coded as follows:

"v"	Viewpoint location (viewing vectors and angles)
"b"	Background color
"l"	Positional light location
"f"	Object material properties
"c"	Cone or cylinder primitive
"s"	Sphere primitive
"p"	Polygon primitive
"pp"	Polygonal patch primitive

See the NFF article for a discussion of each entity.

Output Format

The MTV output format is based on the PPM format, a part of the pbmplus package of utilities developed by Jef Poskanzer. PPM is described in the PBM article in this book, and the utilities are included on the CD-ROM that

Ex.1007 / Page 642 of 1158
APPLE INC.

accompanies this book. The MTV output format differs only trivially from the PPM format. The author of the MTV format describes the output format as follows:

> An MTV format image consists of an ASCII header followed directly by the image data bytes. The ASCII header is merely a string containing the width and height followed by a newline character. The following C statement will print out the ASCII header:
>
> ```
> fprintf(fp, "%d %d\n", width, height) ;
> ```
>
> This is followed directly by the image data, which is written out as three unsigned bytes per pixel, originating at the upper left of the image. This is identical to how the bytes are written out in the PPM image format.
>
> If you desire to write PPM format files, you merely need to change the line which outputs the ASCII header to the following:
>
> ```
> fprintf(fp, "P6\n%d %d\n255\n", width, height) ;
> ```
>
> Here is an example of a small pixmap in this format:
>
> ```
> P3
> # feep.ppm
> 4 4
> 15
>  0  0  0    0  0  0    0  0  0   15  0 15
>  0  0  0    0 15  7    0  0  0    0  0  0
>  0  0  0    0  0  0    0 15  7    0  0  0
> 15  0 15    0  0  0    0  0  0    0  0  0
> ```
>
> Programs that read this format should be as lenient as possible, accepting anything that looks remotely like a pixmap.

The PBM article provides additional information about the PPM output format.

For Further Information

For further information about the MTV format, see the specification included on the CD-ROM that accompanies this book, as well as the specifications for NFF and PBM.

Ex.1007 / Page 643 of 1158
APPLE INC.

The MTV ray tracer is no longer being maintained by Mr. VandeWettering, who considers it dead. However, ample documentation is provided with the format package, should the need ever arise. You may also be able to get additional information from:

Mark VandeWettering
Pixar
1001 West Cutting
Richmond, CA 94804
Voice: 510-236-4000
FAX: 510-236-0388
Email: *markv@pixar.com*

You can also contact:

Tony Apodaca
Email: *aaa@pixar.com*

Ex.1007 / Page 644 of 1158
APPLE INC.

NAME:	NAPLPS
ALSO KNOWN AS:	North American Presentation Layer Protocol Syntax
TYPE:	Graphics Protocol/Metafile
COLORS:	NA
COMPRESSION:	None
MAXIMUM IMAGE SIZE:	NA
MULTIPLE IMAGES PER FILE:	Yes
NUMERICAL FORMAT:	ASCII
ORIGINATOR:	ISO/ANSI/CSA
PLATFORM:	All
SUPPORTING APPLICATIONS:	Videotex services, Prodigy
SPECIFICATION ON CD:	Yes (summary description by third party)
CODE ON CD:	No
IMAGES ON CD:	Yes
SEE ALSO:	None

USAGE: Transfer of graphics information to devices such as modems and terminals.

COMMENTS: Although not strictly a file format, NAPLPS is likely to become the basis of one, because it is in wide use in a rapidly growing segment of the computer communications industry.

Overview

NAPLPS (North American Presentation Layer Protocol Syntax) was designed as an information transfer protocol rather than as a file format. However, because NAPLPS data is occasionally written to disk and saved in file form, it is only a matter of time before an actual format stabilizes. For this reason, we are including summary information about NAPLPS in this book.

NAPLPS is used by a number of Videotex services, is supported by special NAPLPS terminals, and is used by Prodigy, a well-known commercial online service. NAPLPS was designed to extend ASCII to provide efficient transmission of text and picture information. It was specifically designed to provide usable information transfer rates even at 2400 baud. Data is sent as a stream of 7-bit or

Ex.1007 / Page 645 of 1158
APPLE INC.

8-bit ASCII characters to provide maximum compatibility with all ASCII-based operating platforms, network hardware, and network software.

NAPLPS defines line, box, circle, arc, polyline, polygon, spline, bitmaps, and fonts, both in palette and 24-bit color. The coordinate model is right-handed Cartesian, meaning that X and Y coordinates increase toward the upper-right of the screen.

A NAPLPS code sequence begins with the characters ESC 25 41 and ends with the sequence ESC 25 40. NAPLPS code sequences are designed with an eye toward avoiding standard terminal escape sequences such as those provided by VT100 and ANSI. NAPLPS files are basically segments of the NAPLPS data stream redirected to a file. Properly formatted, NAPLPS data files are not unlike uuencoded binary files. Proponents of NAPLPS claim great efficiencies from the encoding scheme. Much of the burden for image reconstruction falls on the rendering application, so a combination of low-bandwidth transmission channels and high-performance workstations would work well with NAPLPS. Unfortunately, the trend in recent years has been for data to be sent in bitmap format across channels of increasing bandwidth, so it's not clear what the future of NAPLPS will be.

A number of terminal programs currently support NAPLPS on both the PC and the Macintosh platforms.

For Further Information

For further information about NAPLPS, see the excellent article by Michael Dillon included on the CD-ROM that accompanies this book. This article provides an overview and some detailed information about NAPLPS. You can also contact Mr. Dillon at:

Michael Dillon
CompuServe: 71532,137
Internet: *mpdillon@halcyon.com*

Also check out the TurBoard NAPLPS BBS homepage at:

http://www.mindspring.com/~crhoads/shawn/turboard/

NAPLPS is formally defined in standards documents available for purchase from the International Standards Organization (ISO), the American National Standards Institute (ANSI), and the Canadian Standards Association (CSA).

Ex.1007 / Page 646 of 1158
APPLE INC.

Note that information contained in the CSA supplement (see below) is not included in the ANSI version of the document.

International Standards Organization (ISO)
1 rue de Varembe
Case Postal 56
CH-1211 Geneva 20 Switzerland
Voice: +41 22 749 01 11
FAX: +41 22 733 34 30

Ask ISO for the NAPLPS specification.

American National Standards Institute (ANSI)
Attn: Sales Department
1430 Broadway
New York NY 10018
Voice: 212-642-4900

Ask ANSI for document number X3.110-1983.

Canadian Standards Association (CSA)
Attn: Sales Group
178 Rexdale Blvd.
Rexdale, Ontario M9W 1R3
Voice: 416-747-4044

Ask CSA for document number T500-1983 and supplement number 1-1991.

Further information about the NAPLPS format can be found in the February, March, April, and May 1983 issues of *Byte* magazine.

Ex.1007 / Page 647 of 1158
APPLE INC.