hierarchy of increasingly specialized types, each of which inherits the more general properties of its parents, while adding specialist properties which can in turn be inherited by child classes that provide further specialization. For example, in a financial application, *current account* and *savings account* specialize the properties and behavior of a generic *bank account*. A *current account* 'is-a' generic *bank account* that has been specialized; so is a *savings account*.

Polymorphism (the ability to take multiple forms) enables objects to respond either as specialized types or as the types from which they inherit, allowing the programmer 'to ignore the differences between similar objects at some times, and to exploit these differences at other times' [Koenig and Moo 1997, p. 35]. Thus in the financial application example, a *current account* can be treated either as a *current account* or as a generic *bank account*.

Encapsulation

Object-oriented languages are strongly influenced by the idea of abstract data types (ADTs). The central idea of an ADT is that it defines a data structure and the operations which may be performed on it [Bishop 1986, p. 4].[16] To use an ADT it is enough to have access to the (public) operations it supports, without requiring any knowledge of its internal structure, and especially without requiring any knowledge of its implementation (that is, the internal data it contains and how it implements the operations it supports).

ADTs are a powerful idea and mark a big step forward in enabling programmers to create their own, user-defined, complex types, having something like equal status with the built-in types of a language. ADTs really belong to the 'data abstraction' revolution (the revolution before the object-oriented revolution), which spawned the Modula-2 language and culminated in the definition of the Ada language.[17] Ada brought ADTs into the mainstream, but C++ is the language that has taken Ada's ideas and made them successful.[18]

Support for ADTs, that is encapsulation, does not itself define a language as object-oriented (Ada is not object-oriented). However, it is a central idea of object-oriented languages. Encapsulation is the most basic pattern an object-oriented system can use. It is also a key programming

[16] For a different view, see [Madsen *et al.* 1993, p. 278] and [Craig 2000, p. 17].

[17] See [Bishop 1986] for a discussion.

[18] For an insight into why, the aside in [Stroustrup 1994, p. 192] about the relative sizes of the Grady Booch component library is illuminating: 125 000 lines of uncommented Ada to 10 000 lines of C++. Ada wasn't much liked by anyone (see the note in [Kamin and Samuel 1990, p. 248] of Tony Hoare's Turing Award lecture remarks). 'What attracted me to C++ had more to do with data abstraction than with object-oriented programming. C++ let me define the characteristics of my data structures and then treat these data structures as "black boxes" when it came time to use them.' [Koenig and Moo 1997, p. 12].

insight, an important step away from a focus solely on algorithm and implementation. In class-based object-oriented languages, encapsulation of objects is provided automatically by the machinery of class definition.[19] In the case of C++, encapsulation of user-defined data types through the mechanism of class definition is probably the key concept of the language.

Classes define objects whose instances are created at run time. Objects hold values and an object's methods provide the means of access to its values, whether to set, update, retrieve or perform more complex operations upon them. An object's methods define the interface that the object exposes or, in Smalltalk terminology, the protocol that it understands. (Terminology varies between languages: Java has interfaces and methods; Smalltalk has protocols and methods; and C++ has interfaces and what are interchangeably called either methods or functions.)

Object-oriented languages also allow objects to be extended to create new objects. In class-based, object-oriented languages, inheritance provides the extension mechanism. (But prototype languages, for example, use a copy-and-modify 'cloning' mechanism to create new objects from old.)

In C++, there is no requirement to follow the logical separation of interface from implementation with a physical separation of code. In contrast, Java formalizes the separation by separating the class declaration from the class definition (implementation). The interface provided by a class for manipulation of instantiated objects of the class is declared in an interface file, with only one class per file.

Inheritance

Inheritance is the mechanism in class-based languages that allows new classes to be defined from existing ones. Not all object-oriented languages are class-based (e.g., there are actor- and prototype-based object-oriented languages[20]), but most are. Therefore while, strictly speaking, inheritance is not universal in object orientation, it is certainly typical.

Inheritance is a parent–child relationship between types, usually called subclassing in Smalltalk and Java (a class is subclassed from a superclass) and derivation in C++ (a class is derived from a base class). Whereas an abstract data type is a black box ([Stroustrup 1994, p72]) which can't be varied or adapted except by redefining it, inheritance provides a mechanism that allows flexible variation of abstract data types, in order to express both the differences and similarities between general types (such as *BigCat*) and their specializations (*Lion* and *Tiger*).

---

[19] [Beaudouin-Lafon 1994, p. 15] says, 'a class is simultaneously a type and a module', where type implies interface and module implies implementation.

[20] Actor languages with an object-oriented flavor include ABCL and Obliq; Self is probably the best known prototype language and is thoroughly object-oriented [Craig 2000].

The key differences in the way that languages approach inheritance are in whether multiple inheritance is supported or not, and in whether the inheritance hierarchy is singly rooted or not. Smalltalk and Java are singly rooted, meaning that there is a single privileged root class from which all other classes ultimately derive and which defines (and implements) a universal set of common class behavior. In both languages, all classes are subclasses of an `Object` class; Eiffel is similar, with all classes derived from the `ANY` class, either implicitly or explicitly. In C++, on the other hand, there is no universal base class: the inheritance hierarchy may have multiple roots. C++ also allows multiple inheritance, so that classes are unconstrained in the number of parent classes from which they may derive. Similarly, Eiffel allows multiple inheritance. Smalltalk allows only single inheritance, that is, a class may only have one parent, while Java allows multiple inheritance of interfaces, but only single inheritance of implementation.

Inheritance is not just additive. It does not just consist of adding new definitions in child classes; it also enables the redefinition in child classes of the existing behavior of parent classes. Typically this is known as overriding, the child overriding the behavior of the parent with its own specialized behavior.

Object-oriented languages typically distinguish between abstract behavior, which defines an interface to an object but which does not provide an implementation, and concrete behavior, which both defines and implements an interface. Abstract behavior is provided by defining abstract methods (in C++, virtual methods). Abstract methods emphasize the point that inheritance relationships are defined by methods, but not their implementations. Classes can also be defined as abstract. Abstract (pure virtual, in C++) classes cannot have instances. In C++, abstract classes provide the mechanism for polymorphism. Child classes are required to implement the abstract methods of a parent.

Inheritance is explicitly a mechanism of class-based languages. Non-class-based object-oriented languages, for example prototype languages, provide equivalent mechanisms based on the idea of cloning new objects from template objects ('prototypes'), to create 'pseudo-classes' of similar objects, rather than true classes, but the purpose is essentially the same [Appel 1998, p. 310].

Polymorphism

Intuitively, the operations that can be performed on a value depend on the type of the value. Adding numbers makes sense and concatenating strings makes sense, but adding strings or concatenating numbers do not make sense, or not in any generally agreed way.

Different programming languages treat the notion of type in different ways. At one extreme, the functional programming world favors complete type-inference systems that amount to full logics (i.e., languages) in their

own right and are completely independent of any physical machine representations of values. At the other extreme, procedural languages such as C, as well as older languages such as Fortran, have type systems which have evolved naturally, and informally, from the physical representation of values in machine memory (bits, bytes, words, long-words, double-words, and so on).

Object-oriented languages fall somewhere between these extremes. Every object in an object-oriented program is really an instance of a fully encapsulated, and possibly user-defined, type. In a class-based language, class definition is the same as type definition. The inheritance relationships between objects are type relationships.

Polymorphism simply means 'having many forms' [Craig 2000, p. 4]. In an object-oriented context, it is often alternatively described as 'dynamic typing'. Polymorphism exploits a simple principle of substitutability: two objects related by inheritance (or an equivalent mechanism) share a common subset of methods. However, the implementation of those methods may differ.

Methods can be invoked on a child object based simply on what we know about its parent. We know that a set of methods is supported, whatever their implementation and whether or not we know what other specializations have been added. Sometimes we only know the parent class of an object and not which specialization we are dealing with. (For example, we may know that we have an event, but not what type of event we have, or that we are dealing with a document, without knowing what kind of document). We therefore know what common methods are supported by the object, whether or not we know what their behavior is, or what other methods are supported. Often we may not even care about the details, for example if we simply want to tell an object to print itself.

At other times, we may explicitly want to use the specialized behavior of the derived object. Polymorphism is the ability of the object to switch between these different behaviors, to appear in the run-time context of a program variously as an instance of the parent object or as the derived object; in other words the ability of an object to behave differently at different points of the program execution.[21]

How polymorphism is implemented varies between languages. For example, Smalltalk uses universal run-time type checking to provide the underlying support for run-time polymorphism. C++, on the other hand, employs static type checking, but allows a 'virtual' dispatch mechanism to support constrained run-time polymorphism.[22]

---

[21] See [Koenig and Moo 1997, p. 77] for a printing example.

[22] Polymorphism is also frequently referred to as 'dynamic binding'. [Bar-David 1993, p. 87] gives a slightly different slant to his definition of dynamic binding as 'the ability of an object to bind – dynamically at run time – a message to an action (or code fragment, if you will). The idea is that, in a given system, many different objects may respond to the same

A weaker notion of polymorphism is usually qualified as parametric polymorphism. It refers to functions which can be applied to arguments of a different type. This is not polymorphism in the same sense as dynamic typing, because the implication is that such functions execute identical code [Appel 1992, p. 7] whatever the argument type; in other words, overriding of implementation is not allowed. A simple example is the language operator (i.e., the built-in function) denoted, in the C language, by &; it creates a pointer to its argument, irrespective of the argument type [Aho *et al.* 1986, p. 364]. Functional languages such as ML and Scheme support parametric polymorphism systematically, while conventional procedural languages such as C and Pascal do not (although they may support occasional instances, such as the & operator in C). Object-oriented languages typically support polymorphism in its stronger sense.

Different languages adopt different strategies for type checking. The primary distinction is between static and dynamic type checking. Static type checking means that types are checked at compile time: if the compiler encounters static type errors, it rejects the program. Dynamic type checking occurs at run time, that is, during program execution: if the program encounters dynamic type errors, it halts the program or flags a run-time error in some other way. A different way of stating the distinction between them is to say that static typing concerns the type of the declaration (for example, a C++ reference to a variable or a C pointer to a variable), while dynamic typing concerns the type of the value (for example, a Smalltalk object) and the difference emphasizes the different underlying programming philosophies.

Statically typed languages include Pascal, C, C++, Ada and the functional languages ML, Scheme and Haskell. Statically typed languages are regarded as strongly typed if the type system enables static analysis to be sufficient to determine that execution of a program will be type correct [Aho *et al.* 1986, p. 343], although it is not required that the compiler necessarily be able to assign a type to every expression. Such expressions require run-time evaluation. Strongly typed languages include Pascal, C, Ada, Java, Simula, Modula-3 and C++ (except for the single case of a dynamically typed method).

Dynamically typed languages are those in which all expressions are typed and checked at run time. For example, Smalltalk and Eiffel use 'dynamic method lookup' [Appel 1992, p. 7]. (Smalltalk is sometimes described as untyped, like Lisp, but it makes more sense to say that the type information has been moved where it belongs, into the object as part of the object's encapsulation).

---

message – say ''print'' (i.e., display yourself to a display device); they just respond differently'. Alternatively, see [Ambler 2004]: 'Different objects can respond to the same message in different ways, enabling objects to interact with one another without knowing their exact type'.

Most languages that perform static analysis (such as Pascal, C, Ada, C++ and Java) require type declarations in programs for all declared types, whether data, operations (i.e. procedures, functions or methods, depending on the language's terminology) or user-defined types. (ML and Haskell are exceptions that use static type inference).

C++ is something of a hybrid. While it mostly checks types statically, it explicitly enables a mechanism for dynamic typing for polymorphic objects, as well as a limited form of type analysis (it is really mangled name matching) for objects loaded at run time, such as precompiled libraries.

Dynamic typing in C++ is enabled by addressing an object through a pointer or reference (although not every pointer or reference implies polymorphism of the object on the other end[23]). A C++ pointer addresses an object of the type of the pointer or of a type publicly derived from it. The type is resolved at run time, in principle at each point of execution in the running program. In C++ (and in Java), this allows the use of a parent class reference to address a local variable, a class instance variable or a method parameter instantiated by an object of a child class. In this case, it is the real type of the object which determines which methods are called, in cases where methods are overridden in a class hierarchy.[24] This enables a program to invoke a method on an object with a single call that is 'right first time', regardless of where in the class hierarchy the object is defined and regardless of the actual behavior of the method. (A `calculateBonus()` method in a payroll system, for example, performs the correct calculation, depending on the real type of the object, not on the type of the pointer.) The alternative, if polymorphism were not available, would require testing for all possible types of the object to isolate the particular case in every case every time, which is laborious and error prone, as well as verbose.

Java is statically typed but all Java methods are bound at run time.[25] All Java objects are typed and their types are known at compile time. The compiler performs static type checking. Dynamic binding means that polymorphism is always available, that is, all methods are like virtual methods in C++ and can be overridden. In other words, every subclass can override methods in its superclass [Niemeyer 2002, p. 11].

Both the static and dynamic approaches have their adherents. The really significant difference between them is that each lends itself to a certain style of programming.

The most common arguments in favor of statically typed languages are run-time efficiency (all types are computed at compile time, so

---

[23] See the discussion in [Lippman 1996, p. 21].

[24] See the discussion in [Warren *et al.* 1999, p. 33–34].

[25] Run time polymorphism, that is, dynamic typing, applies in C++ only through virtual functions [Koenig and Moo 1997, p. 35]. A virtual function counts here as a pointer, i.e. a pointer to a function in some class, its base class or a class derived from it.

there is no run-time overhead) and program safety. Thus, says [Appel 1992], programs run faster and errors are caught early. In statically typed languages, many programming errors are trivial type errors due to programmer oversight, which can be caught and corrected at compile time. In dynamically typed languages, these may arguably become run-time errors. (Arguably, because adherents of dynamically typed languages would probably claim that the rigidity and inflexibility of the type system caused the errors in the first place.)

Type declarations probably do improve code readability and make programmer intentions clearer. On the other hand, dynamically typed languages such as Smalltalk and Python allow greater expressivity and explicitly license a more exploratory programming style, as well as avoiding some of the binary compatibility problems of applications and libraries written in statically typed languages.

4.5 The Languages of Object Orientation

Smalltalk remains the canonical object-oriented language, but almost certainly more object-oriented code has been written in C++ and quite possibly in Java too. These three languages constitute the object-oriented mainstream. Python, a newer language more specialized for scripting and rapid development, may well be on its way to joining them in the mainstream; if it can oust Perl from its position as the universal language of the Web, it will certainly succeed. C# is another, newer language which has set its sights on conquering the Java world as part of Microsoft's .NET services effort. However, it currently remains a niche language.

The differences between these languages and the other object-oriented languages which come and go, are in large part about style (and history). However, in the differences between Smalltalk and C++ in particular, there are insights into more interesting, and deeper, differences about what matters most in programming, for example the trade-off between flexibility and correctness or, perhaps more precisely, what is the best route to correctness and to well-behaved programs which are also capable of evolving to serve the evolving needs of their users. Differences of language style reflect different intuitions about programming style (that is, not just about the style of programs, but also about the different styles of programming practice, the actual activity of designing and writing programs).

The key language differences can be fairly easily summarized:

- single versus multiple inheritance
- a single root class versus ad hoc class hierarchies
- dynamic versus static type checking and method binding.

Some other differences seem to have been relegated to questions of academic interest only by the success of the mainstream languages:

- encapsulation versus delegation
- classes versus prototypes.

Languages which seemed to hold promise for a more concrete and intuitive approach to exploratory programming (for example, Self or Squeak, both Smalltalk derivatives) seem to have been rapidly sidelined.

One seemingly arcane research topic which has migrated in the other direction, from the fringe to the language mainstream, is *reflection* or introspection. Both Java and C# now support reflection, as does Objective-C; run-time program objects are reflective (introspective) and are able to consider themselves as data to be manipulated. Smalltalk also uses reflection, in particular as the mechanism which enables objects to examine themselves to discover their own types.

Java supports reflection for similar reasons, but with a different mechanism, providing a set of reflective classes that allow users to examine objects to obtain information about their interfaces [Craig 2000, p. 197] and to serialize objects. (In Smalltalk, reflection is a meta-property of all class objects.)

Reflection is a rather esoteric property of a few languages (Smalltalk, Self, Java and C#), but it should be seen as part of the search to define more flexible languages, with more natural support for distributed and parallel programming, and part of a longer tradition of languages which include meta-level operations enabling a program to represent itself and describe its own behavior. Smalltalk, like Lisp, can manipulate its own run-time structures [Craig 2000, p. 184].

Other areas of object-oriented research focus less on language techniques than on run-time issues, such as just-in-time compilation techniques (for Java and C#, as well as Python, which are all interpreted languages). It seems unlikely that the familiar object-oriented languages will evolve very radically. The more likely areas of change will be the drive towards binary-object encapsulation for distributed programming (in the style of CORBA), which perhaps suggests an eventual convergence between object-oriented techniques and more declarative programming language styles, under the influence of the success of XML. (Declarative programming supports greater semantic transparency.)

Meanwhile, with C++ and Java, and perhaps Python, as the dominant languages, the programming mainstream now seems very squarely object-oriented.

Smalltalk

Smalltalk dates back to 1972 when the research project from which it originates began, although it came of age with the Smalltalk-80 release.

It drew its inspiration from Simula and was developed by Alan Kay's research group at Xerox PARC [Beaudouin-Lafon 1994, p. 57]. In many ways, Smalltalk is the canonical object-oriented language and it was certainly the first to achieve critical mass. It was launched into the spotlight in 1984, when *Byte* magazine devoted an entire edition to it.

Smalltalk gathered significant commercial momentum. However, since its peak in the late 1980s and early 1990s, it has largely been in decline. It has been decisively beaten (in terms of the programming mainstream) by C++ and Java. Its most interesting legacy has been its promise of a very different way of creating large programs, a more evolutionary and exploratory approach than is encouraged by the 'specification first', top-down style of C++.

Smalltalk is a dynamically typed, class-based, message-passing, pure object-oriented language:

- Everything is an object and every object is an instance of a class.
- Every class is a subclass of another class.
- All object interaction and control is based on exchanges of messages.

Conceptually at least, Smalltalk is remarkably clean and uniform, applying the object approach consistently and deeply. In particular, Smalltalk has a single root class, called `Object`, from which all objects ultimately inherit. `Object` itself inherits from the class named `Class`, which inherits from itself (to satisfy the rule that all classes are subclasses of another class).

In Smalltalk, a class whose instances are themselves classes is called a meta-class. Thus `Class` is an abstract superclass for all meta-classes and every class is automatically made an instance of its own meta-class. This mechanism is used to introduce the notion of *meta-class* methods ('class methods'), which all subclasses inherit and which define the canonical shared class behavior. For example, class methods typically support creation and initialization of instances and initialization of class variables.

The Smalltalk system at run time consists only of objects. All interactions between objects take the form of messages. The message interface of an object is known as its protocol and message selection determines what operations the receiving object should carry out. Each operation is described by a method. There is one method for each selector in the interface of the class.

All objects are run-time instantiations of classes. Classes are defined by class descriptions that specify the class methods (i.e. the meta-class methods), instance methods and any instance variables [Goldberg and Robson 1989, p. 79]. Method specifications consist of a message pattern (equivalent to a function prototype in C++) which specifies the message

selector and argument names, and an implementation [Goldberg and Robson 1989, p. 49]. A protocol description for each class lists the messages understood by instances of the class.

The message-passing model is uniformly applied as the single control mechanism for objects. Objects respond to messages and issue messages, and there is no other control mechanism in the system. For example, a new object is created by sending a message to the required class, which is itself an object (because `Class` is itself an object) and can therefore receive messages. The class object creates the new class instance. Message expressions specify a receiver (the intended target object), a selector (the message name) and any arguments [Goldberg and Robson 1989, p. 25].

Inheritance is used as the mechanism which enables sharing between classes. In other object-oriented languages, classes are definitional constructs that define instances and it is these instances which are objects (i.e., an object instantiates a class but a class is not itself an object). This is not the case in Smalltalk, in which everything is an object, including numbers, characters, Booleans, arrays, control blocks, and even methods and classes. Smalltalk has a rich hierarchy of ready-made classes (230 classes in Smalltalk-80 with 4500 methods) [Mevel and Gueguen, p. 5].

The object purity of Smalltalk extends all the way down to what in other languages would be the purely syntactic level of control structures. This makes its syntax idiosyncratic compared with other languages. Probably the most unfamiliar aspect of Smalltalk syntax for anyone with a background in procedural languages is the absence of familiar control constructs such as `if-then-else`. Instead, control blocks act as switches. For example, compare a conventional C-style `if-then-else` with a Smalltalk conditional block, using a Boolean object and `ifTrue:` and `ifFalse:` messages. Certainly it can appear radically unfamiliar for anyone coming from a more conventional programming background.

A final idiosyncrasy (although it may seem more natural to newer generations of programmers brought up on IDEs rather than the command-line) is that Smalltalk cannot be invoked as a simple language interpreter or compiler, but is instead part of a complete graphical programming environment. Smalltalk programs do not compile into conventional executables and libraries, with conventional linkage models, but instead dynamically update the running image of the complete live environment. The Smalltalk system can thus be modified at run time (unlike a conventional compiled executable). The language (and its associated tools) are thus embedded in a live, interactive environment, which is consistent with the origins of the language and its goals (a teaching language for novice programmers, based on a 'physical world' metaphor). Snapshots of the environment can be created as persistent images.

An irony is that where Smalltalk aims for simplicity, the language (and the associated 'object theory') turns out to be surprisingly complex. While Smalltalk failed to gain much hold as a teaching environment, it

found a number of commercial niches (it remains popular for financial modeling applications) and it has retained its place as an 'extreme' language for (far from novice) object purists. A great deal of advanced object-oriented programming practice and theory, from patterns to the philosophy of reflection to extreme programming praxis, have originated in the Smalltalk world.

An interesting Smalltalk spin-off is the Self language, designed by Randall Smith and David Ungar, which originated at Xerox as a vehicle for exploratory programming and an experiment in an object-oriented language not based on classes. Instead of classes, Self is based on the notion of prototypes. New objects are derived by cloning and modifying existing prototype objects. Self takes the idea of a language embedded in an environment modifiable at run time to an (interesting) extreme. By removing both the theory and the machinery that comes with classes (inheritance, polymorphism and so on), it removes almost all of the complexity, while still retaining the power of object-based abstraction. Self espouses as a central principle that an object is completely defined by its behavior.[26] The corollary is that programs are not sensitive to the internal representations chosen by objects or, indeed, any other hidden properties of programs.

C++

C++ originated from a networking-support research project at Bell Labs in 1979, as an attempt by Bjarne Stroustrup to improve C by adding class concepts derived from Simula to support powerful but type-safe abstract data type (ADT) facilities. Indeed those origins are made transparent by its first incarnation as 'C with classes'.

The central concept of C++ is that of class [Koenig and Moo 1997]. Classes enable user-defined types that encapsulate behavior and data. Originally, C++ classes began as elaborations of C structs. While structs allow structured data to be defined and managed to create user-defined complex data types, classes extend the idea to include method definitions as well as data. (C++ retains the notion that a simple class that defines no methods is synonymous with a struct.)

In its first implementations, C-with-classes and later C++ were implemented as pre-processors, which translated C++ into plain C and then invoked the standard C compiler. (Again, the history is in the name: the first C++ implementation was named Cpre) [Stroustrup 1994, p. 27]. In a general sense, C++ thus includes the C language but C++ is not a pure C superset (unlike Objective-C, for example).

The goal of C++ is to enable the same level of type safety as is enjoyed by built-in language types to be enjoyed by user-defined data types. C++

[26] See the *Self Programmers Reference Manual*, p. 55 at ***http://research.sun.com/self/language.html***.

also provides improved type safety for built-in types compared with C, for example with language constructs designed to support immutable values (the `const` construction and the 'reference' operator). Its secondary goal is to do so without compromising either the efficiency of C or C's ability to operate (when necessary) close to the machine level.

Compared with Smalltalk, its goals make C++ inherently a hybrid language, sacrificing purity in favor of pragmatism. C++ is often said to be not an object-oriented language at all, but a language which can be used to program in a number of different styles. The more use that is made of advanced language features, the closer the style becomes to object orientation. However, there are advanced features which have little to do with object orientation (as understood in the purer sense of Smalltalk at any rate), for example the templating support for parametric (also known as generic) programming styles.

In summary, C++ is a strongly statically typed language with support for classes.

- Objects are optional, but when used they are based on classes, which, at one extreme, may be C-like structs and, at the other, may define pure virtual (polymorphic) objects or may fall somewhere between.
- Objects are created from classes by constructor methods and are deleted by destructor methods, which may be defined (for complex classes) or default to standard methods if not defined.
- Objects can control access to their data and methods by declaring them private, shared or public.
- Values can be made immutable by declaring them `const` and all objects can be passed by value, by reference or by pointer.
- Separation of interface from implementation is encouraged but not enforced (for example, methods may be declared inline and their implementation specified at the point of definition, within a class definition). Definition and implementation can be mixed and there is no requirement for separate interface definition files. Multiple definitions and implementation specifications can be provided within a single file. C-style `#include` preprocessor directives are used to manage definition inclusion.
- There is no notion of a root class and, therefore, no definite class hierarchy. Multiple hierarchies can be created within a single program and multiple inheritance is allowed (so that a single class may inherit from multiple parents). Advanced object-oriented features such as reflection are not supported.

Run-time polymorphism is the exception in C++ rather than the rule and, for all other cases, type checking is performed at compile time.

Run-time polymorphism is enabled only for classes which are defined as pure virtual, in which case method dispatch is completed at run time through a 'vtable' (a virtual method dispatch table). Virtual methods use run-time binding and are not determined at compile time.

C++ retains a conventional C-style execution and linkage model. There is no automatic garbage collection in C++. Memory management is the responsibility of the programmer, making the language flexible and powerful but also dangerous (carelessness leads to memory leaks).

Java

Just as C++ began as an exercise to improve C, so Java began as an exercise to improve C++ and, in particular, to simplify it, straighten out inconsistencies and make it less dangerous (for example, proof against memory leaks) as well as more secure (in the sense of tamper-proof, the origin of the Java 'sandbox' application model) and, therefore, more suitable for a wider range of devices (in particular, for smaller, consumer-oriented systems). Perhaps even more importantly, from the beginning the Java implementation model aimed at maximum platform neutrality and a write-once–run-anywhere model.

Java language programs are thus compiled into an interpreted intermediate language which is executed by a Java virtual machine (VM) running on the target hardware. Any Java code runs on any Java VM, thus providing abstraction from physical hardware. In other words, Java provides a software environment for code execution rather than a hardware environment.

In this sense, Java is like the pure object-oriented model of Smalltalk, which similarly provides a software execution environment based on a VM. Unlike Smalltalk, Java programs are separable from the execution environment and its linkage model is more akin to a conventional executable and library-linkage model.

The VM approach also allows Java to meet its goals of robustness and security. The VM controls access to the resources of the native environment, thus enabling a garbage-collected execution environment (so that memory management is the responsibility of the environment, not the program), as well as a security sandbox, isolating Java programs from the native environment (malicious software can at worst only attack other code executing on the VM and has no access to the VM itself, nor to the underlying system).

Java programs pay a price for the execution model, in the overhead of interpreting Java intermediate byte code. However, Java VM technology exploiting sophisticated compilation techniques has eroded the raw speed differences between executing Java byte code on a VM and executing native processor instructions, to the point where execution speed differences are almost insignificant.

Java has been less successful at reducing latency of program startup, however, which requires the complete Java environment to be initialized. Java has also struggled to slim down its substantial platform memory footprint. For desktop PCs and 'single-function' consumer devices such as set-top boxes, this is less of an issue than it is, for example, on mobile phones, where Java competes for resources with native code. Pure Java solutions such as JavaOS, which replaces the native operating system with a lightweight Java system sufficient only to host the VM, have not been successful to date, although the Jazelle project has challenged conventional solutions by providing a Java solution in dedicated hardware. Jazelle remains a contender in the mobile phone space.

From a language perspective, Java makes an interesting contrast with C++. It succeeds in its goals of providing an object-oriented language that is simpler and purer than C++, while avoiding the syntactic eccentricities of Smalltalk; it remains syntactically quite conventional and close to its C++ origins.

Like C++, Java is strongly statically typed. Unlike C++ and like Smalltalk, it is a purely class-based language, with an `Object` root class.

- Native number, character and string types are defined by the language; all other types (including all user-defined types) are objects.
- Every object is an instance of a class and every class is a subclass of another class, except for the root class.
- Objects are created from classes by constructor methods and are deleted by destructor methods, which may be defined (for complex classes) or default to standard methods if not defined.
- Objects can control access to their data and method members by declaring them private, shared or public.
- Values can be made immutable by declaring them `const`, and all objects can be passed by value, by reference or by pointer.
- Separation of interface from implementation is enforced. Every class consists of an interface definition and an implementation specification in separate files, with only one class per file.
- Unlike C++, all objects are run-time polymorphic (all methods employ late binding).
- Garbage collection is automatic. All program resources are cleaned up and recovered by the VM when a program completes (or is terminated).

Java's success has been striking and, in many ways, it is a model language. However, compared with C++, it is relatively inflexible and

constrained (by design) and its deliberate isolation from the underlying device makes it generally unsuitable as a system-level language.

Microsoft has made its own attempt at improving Java and providing a managed-code solution of its own (for the .NET services platform, which competes with Java) in the form of C#. As a language, C# contains some interesting features, including a reflection model. However, the history of C#, which first emerged as a set of unilateral Java extensions, makes it somewhat unconvincing as a genuine language advance.

Other Languages: Objective-C, Eiffel and Modula-3

Objective-C was written by Brad Cox in the early 1980s. It has a visible Smalltalk influence, for example in some of its syntax, and in its adoption of run-time typing (in contrast to C, C++ and Java). Also unlike C++, it is a true superset of ANSI C, that is, it is a pure extension of C that leaves the core of C unrefined.

It was adopted for the NeXTStep, which employed a Mac-based flavor of Unix, and from there it was inherited by Mac OS X, in which it remains highly visible. (For native application development, the object hierarchy remains based on Objective-C, complete with the NeXTStep, i.e. NS, class-naming convention.) Objective-C was also an explicit influence and, indeed, the inspiration and model, for the Psion in-house object-flavored C that preceded the adoption of C++ for what became Symbian OS.

Eiffel emerged at around the same time as Objective-C, that is, after Smalltalk but before C++ had become dominant. Eiffel was designed as a commercial, pure object-oriented language intended to compete with Smalltalk, with a more conventional syntax. It included a comprehensive and pure object-oriented class library, including ready-to-go container, collection and iterator classes, well in advance of anything comparable in the C++ world. (The C++ Standard Template Library emerged well after the C++ language.)

In the Pascal lineage, Modula-3 evolved by way of Modula-2, adding object-oriented features and garbage collection.[27] Both Eiffel and Modula-3 are influenced by Simula, but while Simula and C++ allow a choice between static and dynamic binding, with dynamic binding provided via virtual methods, Eiffel and Modula-3 offer a pure polymorphic model with universal dynamic typing and run-time binding, for which run-time efficiency is the trade-off.

In other respects, both languages share similarities with C++. Classes in these languages are elaborations of the concept of a record, a description of a list of fields together with the methods that operate on them (just as C++ classes are elaborations of the concept of a C struct; structs and records are, in essence, synonymous). Again like C++, both Eiffel and Modula-3 allow multiple inheritance.

[27] According to ***www.m3.org***, the language was first defined in 1989.

Part 2

The Layered Architecture View

5

The Symbian OS Layered Model

5.1 Introduction

This book explains the architecture of Symbian OS using the system model (see the fold-out and Figure 5.1), which represents the operating system as a series of logical layers with the Application Services and UI Framework layers at the top, the Kernel Services and Hardware Interface layer at the bottom, sandwiching a 'middleware' layer of extended OS Services.

In a finished product, for example a phone, Symbian OS provides the software core on top of which a third-party-supplied 'variant' user interface (UI) provides the custom GUI with which end-users interact and which directly supports applications. Typically, the variant user interface, including the custom applications supplied by the phone manufacturer, is considerably bigger than Symbian OS itself.

Beneath the operating system, a relatively small amount of custom, device-specific code (consisting of device drivers and so on), insulates Symbian OS from the actual device hardware.

5.2 Basic Concepts

The remainder of this chapter summarizes the key concepts of the system model and then describes the operating system layer by layer, starting at the top with the UI Framework and working down to the Kernel Services and Hardware Interface layer.

The basic approach taken by the model is to decompose the operating system into layers, and to further decompose the layers as necessary into blocks and sub-blocks before finally arriving at collections of individual components. Layers are the highest level abstraction in the model; components are the lowest level abstraction, the fundamental

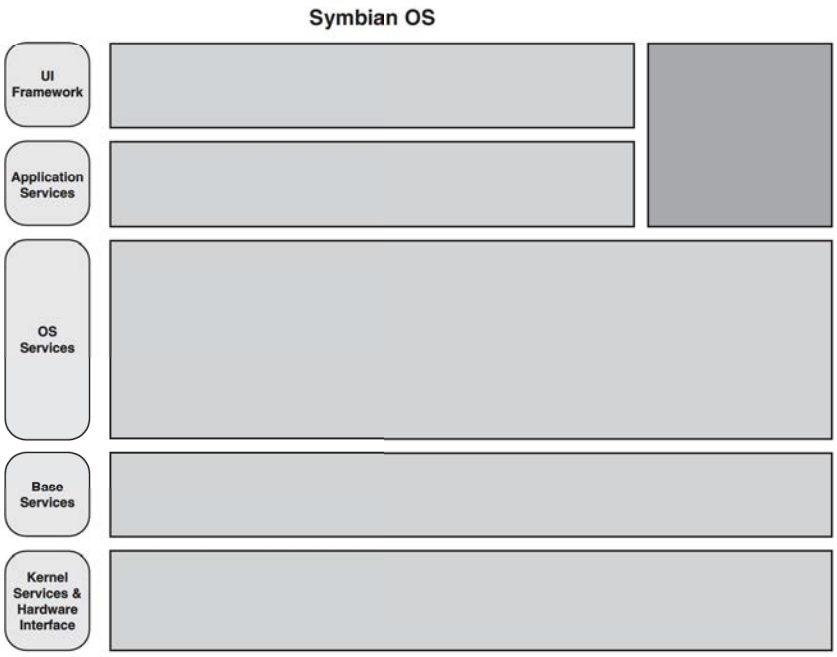


Figure 5.1 Symbian OS layered system model

units of the model; blocks and sub-blocks decompose layers by functionality – roughly speaking, by broad technology area. The key concepts used by the system model therefore are layers, blocks and sub-blocks, component collections and components.

Components provide the essential mapping from the logical model to the concrete system. While layers, blocks and sub-blocks are essentially logical concepts, components are physically realized in software, typically consisting of multiple files in the operating system delivery (e.g. source code files including test code; built executables including libraries; data and configuration files; build files; and documentation). However, from the perspective of the model, components are treated atomically and constitute the smallest units of architectural interest.

The complete component set shown in any particular version of the model represents the superset of all components delivered by that release of the operating system and intended to run on any Symbian OS device, whether a phone or some other product, a development board or other test hardware, or an emulator build of the system running on a host operating system (such as Microsoft Windows).

Test components and tools are considered outside the scope of the Symbian OS model, although they form an essential part of the model of the complete delivery of the operating system as shipped by Symbian

to customers. (They are shown in a full product model as the Symbian Toolkit.)

Because the model reflects the concrete system, a new version of the model is published for each release of Symbian OS. The model has also evolved in its own right since the first versions were published for Symbian OS v7, in particular to bring it closer to the concrete system.

Layers

The model adopts a conventional software architecture interpretation of layers [Buschmann *et al.* 1996]: each layer abstracts the functionality of the layer beneath and provides services to the layer above.

Within each layer, components are either grouped directly into collections according to functionality (and to some extent also according to collaborations and shared dependencies); or are grouped into collections within blocks and possibly sub-blocks, which are broadly based on technologies.

The goal of the model is to impose manageable granularity onto the operating-system architecture, to make it easier to understand and to navigate. Hence, layers are useful approximations of structure, not precise specifications of architectural relationships. There is no concrete mechanism that instantiates layers in the existing system (i.e. there is no make file or equivalent).

However, the broad principles of the layering hold good: although there are some exceptions, dependencies in general flow downwards from higher layers to lower layers; and dependencies in the reverse direction are considered to be anomalies. In general, services are abstracted through the layers, with higher layers abstracting the services of lower layers, although for reasons of efficiency there is no requirement that layers only access the services of the layer immediately below them; thus the functionality of lower layers is accessible to all layers above.

One reason for showing the system as layered is to show how system functionality is increasingly abstracted away from hardware (at the bottom) and towards users (at the top); successive groups of tasks are increasingly abstracted from more basic tasks. A widely accepted principle for creating a layered model of a system is the 'inverted pyramid of reuse', characterized by the slogan 'Keep the base layer slim' [Buschmann *et al.* 1996, p. 39].[1]

Layers in the system model are defined with the following guidelines in mind:

[1] 'Layers' is a well known architectural pattern, the best known example probably being the OSI Seven-Layer Model. The Layers pattern is described and discussed in [Buschmann *et al.* 1996].

- all the services provided by a layer are at a similar level of abstraction
- a layer is relatively logically cohesive and relatively self-contained (both inexact terms, used with commonsense meaning)[2]
- a layer provides services to higher layers ('upwards')
- a layer delegates tasks to lower layers ('downwards')
- dependencies flow consistently from higher layers to lower layers (but dependencies are allowed sideways within layers)
- requests travel downwards
- notifications travel upwards
- higher layers abstract the services of lower layers away from machine-centric services towards user-visible functionality
- a layer provides services as far as possible via well-defined external interfaces, which can be separated from the internal interfaces available within the layer
- a layer could be a delivery unit (although, in the current system, no layer is delivered independently).

Blocks

A block or sub-block in the system model (see Figure 5.2) roughly corresponds to a 'technology domain'.

Blocks are used as a pragmatic way of partitioning layers into meaningful divisions according to commonsense criteria, with sub-blocks providing finer grained divisions for convenience. There is no concrete mechanism that instantiates blocks or sub-blocks in the existing system (i.e. there is no make file or equivalent).

Blocks in the system model are defined with the following guidelines in mind:

- a block is relatively logically cohesive and relatively self-contained
- a block is relatively cohesive and relatively decoupled (measured in terms of the coupling of the component collections it contains)
- a block provides services to blocks in the same layer ('sideways') or to blocks or component collections in higher layers ('upwards')
- a block delegates tasks downwards or sideways

[2] Cohesion and coupling are standard concepts used to analyze software complexity. See, for example, [Henderson-Sellers 1996] and the influential papers by Lionel Briand and others at the Fraunhofer Institute, such as [Briand *et al.* 1997].

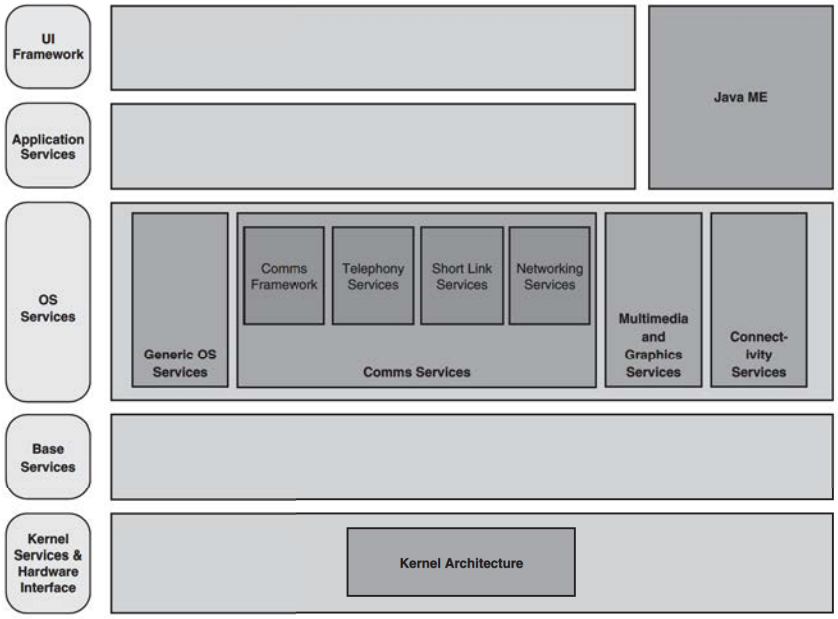


Figure 5.2 Block decomposition in the system model

- a block 'consolidates' the sum of services provided by the component collections it contains into a technology domain
- a block is not a delivery unit – it makes sense to partially deliver, update or remove a block.

Components and Component Collections

Components are the basic entities of the model and the smallest units of architectural interest. Importantly, components have a concrete interpretation in the source system, corresponding approximately to parts of the source tree controlled by a single high-level build file (an MRP file in the Symbian build and delivery idiom but, more generally speaking, a high-level make file).

Components are also the basic units of optionality in the system, the level at which common, optional and replaceable functionality is defined and at which it may be (respectively) included, removed or re-implemented by the respective licensees of Symbian OS.[3]

Component collections group individual components into coherent sets of collaborating components. In principle, a component collection

[3] For a more detailed discussion, refer to Appendix A.

delivers a complete, discrete and identifiable subset of system functionality. In practice, component collections are derived from a 'commonsense' analysis of existing system functionality, as well as the physical organization of the source tree.

There is no concrete mechanism that instantiates component collections in the existing system (i.e. there is no make file or equivalent).

Component definition follows these principles:

- a component is the smallest architectural unit of the system
- a component is understood as a set of implementation units that are built together to provide a discrete, reusable piece of the system
- in concrete terms, a component is identified with a single MRP file that ensures alignment with build and delivery mechanisms (in versions of the model up to Symbian OS v8, a component is identified with a high-level `bld.inf` file rather than an MRP file.)

Components should also obey the following guidelines and display these properties:

- a component is relatively cohesive (in essence it has been designed as a discrete part of the system)
- a component is a reusable unit of the system
- a component is the smallest unit of architectural significance and the finest grained unit of description, management and distribution of the system
- a component is implemented by at least one and possibly many collaborating sub-units
- no part of any component is shared by other components
- all interfaces defined at higher levels of the model are implemented by components.

In all, the system model for Symbian OS v9.3, the latest version of the operating system at the time of writing, defines approximately 250 individual components.[4] However, there is still a significant degree of idealization in the component definitions and, in many cases, the detailed mapping from the model to the system as built and delivered is approximate. In other words, the model serves as a useful logical

[4] Appendix A documents 258 components, for example, and does not include Toolkit components.

description, but cannot necessarily be unambiguously followed down to file level. (Improving alignment is an ongoing task.)

Component collections are defined with the following guidelines in mind:

- a component collection is relatively cohesive and relatively decoupled (in terms of the coupling of the components it collects)
- a component collection provides services to other collections within its block or layer ('sideways') or to blocks or component collections in higher layers ('upwards')
- a component collection delegates tasks downwards or sideways
- a component collection groups logically related functionality
- a component collection exposes the interfaces provided by the components it collects
- no component collection is shared between blocks or layers
- no component is shared between component collections
- a component collection is not a delivery unit – its individual components may be delivered, updated, or removed singly.

5.3 Layer-by-Layer Summary of the Symbian OS v9.3 Model

A high-level view of the system model for Symbian OS v9.3 is included in this book as a fold-out diagram.

All releases of the operating system from Symbian OS v7.0 to Symbian OS v9.3 share the same layer decomposition.

- UI Framework layer: The topmost layer of Symbian OS provides the frameworks and libraries for constructing a user interface, including the basic class hierarchies for user interface controls, and other frameworks and utilities, including concrete widget classes used by interface components.
- Application Services layer: This layer provides support independent of the user interface for applications on Symbian OS. These services divide into three broad groupings:
    - system-level services, such as basic application frameworks, used by all applications

- services providing technology-specific logic, such as messaging and multimedia protocols, that are used by multiple classes of application
- services that support specific individual applications, such as personal information management (PIM) and office applications.

Also included are a number of application engines that are used and extended by a licensee.

- Java ME: In effect, Java spans the UI Framework and Application Services layers, abstracting (as well as implementing) elements of both for Java applications. Symbian's Java implementation is based on Java ME MIDP 2.0 and CLDC 1.1. Java support has been included in Symbian OS from the beginning, but the early Java system was based on Personal Java and JavaPhone. A standard system based on Java ME first appeared in Symbian OS v7.0s. Since Symbian OS v8, the Java VM has been a port of Sun's CLDC HI.
- OS Services layer: The 'middleware' layer of Symbian OS provides the servers, frameworks and libraries that extend the bare system into a complete operating system. The services are divided into four major blocks that provide all technology-specific but application-independent services:
  - generic operating system services
  - communications services
  - multimedia and graphics services
  - connectivity services.
- Base Services layer: The foundational layer of Symbian OS provides the lowest level of user-side services, depending only on the operating system kernel (and related components), which it extends into a useable (but minimal) system. In particular, no services higher than those in the Base Services layer are required for a minimal base port to new hardware (in other words, a minimal base port requires only the two lowest layers of the system).
- Kernel Services and Hardware Interface layer: The lowest layer of Symbian OS contains the operating system kernel itself and supporting components that abstract the interfaces to the underlying hardware, including logical and physical device drivers and variant support that implements pre-packaged support for the reference hardware platforms. Releases up to Symbian OS v8 use the original Symbian OS kernel, Kernel Architecture 1 (EKA1 kernel). In Symbian OS v8.1b and from Symbian OS v9, all systems are based on the new Kernel Architecture 2 (EKA2) real-time kernel.

5.4 What the Model Does Not Show

The System Model shows a static view of the system, in effect a source view based on architectural relationships and abstracted from the details of what code appears in which files. It is not, therefore, a source tree or repository view.

The model also reflects only static (i.e. build-time) dependencies. It does not model processes, the memory contexts that are created on a device when the operating system runs, the threads that run within those memory contexts, or the services that those threads provide.

5.5 History

The system model was first published internally in 2004 (and therefore somewhat after the fact), as a description of Symbian OS v7.0. It was almost immediately updated for Symbian OS v7.0s. That model was first published for a wider audience in [Harrison 2004].

Since then, a revision of the model has been published for each release of the operating system. Since Symbian OS v9, the model has been used in the broader design and specification processes that are part of all operating system releases, providing a design base for each release and supplying the build definition to the software build system.

6

The UI Framework Layer

6.1 Introduction

The UI Framework layer is the topmost layer of Symbian OS (see Figure 6.1) and the immediate interface to the 'variant' user interface supplied by the manufacturer on a phone.

Symbian OS is delivered to licensees in a 'headless' configuration, with a minimal test user interface which is neither complete nor of production quality. (Known as TechView, it is considered to be a test and validation



Figure 6.1 UI Framework layer in the system model

tool and is not, therefore, part of the operating system proper, although in the past it has been exposed to developers through 'preview' SDKs.)

Mobile phone manufacturers who license Symbian OS either replace the test user interface with a production quality user interface of their own, or license a suitable variant user interface. Typically in the latter case, the user interface is pre-integrated and pre-tested with Symbian OS, to simplify the task of bringing a device to market.

Currently two user interfaces are available for licensing: S60 (from Nokia) and UIQ (from UIQ Technology AB). Another important user interface is the MOAP user interface developed in Japan by DoCoMo's FOMA consortium of handset vendors, and used by consortium vendors on FOMA phones.

- **S60** is developed and licensed by Nokia. It ships on Nokia phones based on Symbian OS. Lenovo, LG and Samsung, among others, license and ship S60 phones based on Symbian OS. Licensees have also included Panasonic, Sendo and Siemens.
- **UIQ** is developed and licensed by UIQ Technology AB (until recently a fully-owned, Swedish-based Symbian subsidiary, now acquired by Sony Ericsson). Sony Ericsson, Motorola and Arima license and ship UIQ phones based on Symbian OS.
- **MOAP** is developed by the FOMA consortium in Japan as part of the DoCoMo common software platform for 3G FOMA handsets. FOMA members, including Fujitsu, Mitsubishi, Sony Ericsson and Sharp, ship MOAP phones based on Symbian OS.
- **Series 80 and Series 90** were developed by Nokia but are not licensed to other phone vendors. Series 80 was found on the Nokia Communicator family of devices based on Symbian OS. Series 90 can still be found on the Nokia 7710 phone, but has been merged with S60 for future devices.[1]

6.2 Purpose

The UI Framework layer is the foundation for building customized user interfaces on top of Symbian OS and is the immediate interface between Symbian OS and the variant UI layer.

The UI Framework layer provides the frameworks which custom user interfaces extend, the class hierarchies from which controls specific to

[1] Interestingly, Series 90 began life as the Hildon user interface, developed by London-based Mobile Innovation, now part of Macromedia. Hildon has since been ported as a widget set to the GNU GTK+ user interface toolkit, in which form it appears on Nokia's Linux-based 770 Internet Tablet.

the user interface are derived, and additional supporting components used primarily by user interfaces. It provides some specialist generic frameworks, for example animation, which are used by user interfaces but which are also available directly to applications.

The basic graphical and behavioral user interface abstractions are encapsulated in UI Framework layer components such as the control hierarchy, window interactions and graphics contexts, which determine basic application behavior.

The UI Framework layer is also used by the Java implementation, although Java also makes quite heavy direct use of some lower-level graphics frameworks. (Any such dependencies are transparent to Java applications, which see only Java APIs.)

6.3 Design Goals

The UI Framework layer is intended to enable user interface differentiation without fragmentation. This requires balancing the sometimes conflicting goals of providing a common, consistent functional and behavioral core to all user interface variants in order to provide a consistent development target for application writers while also providing the greatest possible flexibility and customizability to enable maximum user-interface differentiation for phone vendors

The design goals are that:

- the system should be a platform
- the different user interfaces should be distinct platforms.

6.4 Overview

Conceptually, the UI Framework layer has become thinner (functionality has migrated upwards) as the user interfaces built on top of it have become larger and richer; rich user-interface functionality is overwhelmingly in the user-interface variants.

However, important core user-interface functionality is retained in the framework base classes, from which the user interface variants derive. The framework approach means that many of the key user-interface design decisions (the basic user-interface architecture and broad division of responsibilities, managing input methods, the way user interfaces are customized, the basic control hierarchies, and some of the basic GUI application architecture and behavior) are encapsulated in the frameworks, which 'plumb in' the underlying operating system support for event handling (including all input–output events), window management and drawing, font and graphics support, and so on.

- The Uikon framework provides abstracted (i.e. high level or generic), customizable control of the overall GUI look and feel and encapsulates the main classes used to create applications. The underlying implementation of the generic application architecture is provided by lower-level frameworks, such as the Application Services layer.
- The Control Environment hierarchy (widely known as CONE) provides generic screen controls ('widgets') that are free of a look and feel and policies.
- FEP Base, the front-end processor framework, provides input-event capture (by key, pen or voice) and support for language preprocessing engines, for example, for handwriting recognition and exotic script input.

Supporting components provide additional graphical and other utilities (font, color and drawing support for user interfaces, including graphics effects such as fading and animation), as well as some useful frameworks that are used by both user interfaces and applications:

- The UI Graphic Utilities and Graphics Effects components contain common, general-purpose utilities used by user interfaces, for example drawing window borders and fading effects.
- The Animation and BMP Animation components provide frameworks for window animation and bitmap-based and sprite-based animation including animated clocks and animated user-interface elements.
- The Grid framework is a legacy framework specifically supporting cell-like (spreadsheet-style) layout.

Additional support for user-interface customization is included as part of the toolkit delivery (outside the scope of this book), which provides components that customizers may choose to re-implement as part of the variant user interface.

6.5 Architecture

Uikon and Control Environment (CONE) are the two most significant components in the UI Framework layer from an architectural point of view, since they determine the overall user interface architecture. Both also provide essential application support.

For most purposes, applications do not use Uikon directly, but instead use a Uikon-derived custom framework specific to the user interface (for example, Avkon in S60 and Qikon in UIQ). However, there are

exceptions in which applications directly use Uikon; for example, applications directly use the many useful static methods of the user interface Environment object of class `CEikonEnv`.

The Control Environment is used both directly and indirectly by applications. Frequently the main application view is derived directly (from `CCoeControl` or `MCoeView`), bringing all the flexibility of the generic user interface control framework directly to it. Indirectly, applications use the Control Environment through the custom framework or the custom control set of the user interface variant.

Uikon

Uikon can be thought of as the common core on top of which are built the variant user interfaces that actually appear on phones.

Uikon provides a framework for creating user interfaces including the base classes which interface to lower-level system services such as application launching; key mapping and command handling; alarms and notifications; and graphics services.

Uikon supplies the base classes from which user interface variants derive essential application classes (Application, Document and AppUI) and encapsulates the relationships between them.

Uikon supplies the factory classes used by the user interface variant to create the hierarchy of custom concrete user interface control classes, including list boxes, scroll bars, buttons, dialogs and popups. (Basic menus are not controls but windows.)

Uikon loads a static library implementation (interface defined by the UI Look and Feel component) of the core library look-and-feel (LAF) component, which is supplied by the user interface variant. The UI Look and Feel component defines a standard set of methods which the variant user interface implements to define the concrete behavior of user interface elements, for example, layout and behavior of windows; choice of fonts and bitmaps; default location of resource files; system font and text rendering defaults; and the look and feel of toolbar, dialog, button and button container classes. The Uikon Error Resolver Plug-in is a small component that is used by the user interface variant to map system error codes to localized strings. Strictly speaking it is not a plug-in, but a resource file which is built as a dummy DLL.

Uikon provides a server stub which is run to launch other servers expected by the framework or by applications (the alarm server, notifier server, and server-side support for user-interface status panes) and to load implementations specific to a user interface variant for password and alarm notifications. (The Notifier is run inside the Uikon server thread to ensure that memory is always preallocated for those notification dialogs which must never fail, for example the 'Out of memory' dialog itself.)

Uikon provides servers to manage backup and shutdown (used to close running applications when the user starts a backup, and to handle

shutdown when the user switches off the device). In earlier releases of the operating system (up to Symbian OS v7.0) Uikon also supplied a core library of concrete controls and dialogs, `EikCoeControl`; these are now supplied in the customization toolkit, and may be selectively re-implemented by the user interface variant or discarded.

The Control Environment (CONE)

Controls in Symbian OS are window-using, possibly nested, rectangular screen areas that accept user input and other events. (Windows do not necessarily own any controls; menus and sprites, for example, do not.)

Events (such as redraw events, standard events and foreground – 'focus' – events) are supplied by the Window Server to the Control Environment framework.[2] Of these, key, pointer and draw events are routed by the Control Environment to controls. Additional events may be generated by controls themselves, including change of focus events between controls. In effect, controls bring together:

- screen and window behavior as controlled by drawing, redrawing and other events
- graphics states, for example, color, font, brush and other settable attributes
- user-input handling (the Window Server serializes system events, such as key presses and pen taps, and delivers them to the currently active control of the foreground application).

The Control Environment defines the base classes that encapsulate these basic behaviors and the relationship between controls and their environment and define abstract controls. Applications can derive their own types of controls directly or use derived classes provided by Uikon and the user interface variant. The Control Environment, in effect, is the abstract middle layer between the low-level windowing functionality provided by the Window Server and the concrete user-interface classes provided by Uikon and libraries specific to the user interface variant.

- The `CCoeControl` class is the base class for derived controls.
- The `CCoeEnv` class encapsulates the application session with the Window Server, as well as providing utilities to manipulate the graphics state and for other system interactions (for example, it creates an application session with the File Server). Every application owns a singleton object of this class derived from `CEikonEnv` (which is

[2] Note that Window Server focus events are not the same as 'focus events' as understood by controls.

implemented as an active object responsible for routing input-event messages from the Window Server to the application framework `AppUi` class). Typically, the object is accessed from the application framework classes through the derived `CEikonEnv` class. From an application control, the object's methods are accessed through the control's `iCoeEnv` member.

The Control Environment also defines the user interface base class `CCoeAppUi`, providing the application user interface framework (brokered to applications via Uikon and the user interface variant) that manages input events. Key events are managed in the context of the stack of application controls (assigning a key event to the appropriate control).

Front-End Processor Framework

The Front-End Processor (FEP) Framework provides the abstractions that implement user-input capture and preprocessing, for example for handwriting recognition or multitap input systems, in order to capture, process and map user input events onto standard key events.

The FEP Framework provides the base classes for creating FEPs and defines the plug-in interface. The FEP Framework extends Control Environment base classes and is implemented as a DLL that is statically linked to by code which wants to derive from it. The Control Environment manages the creation, ownership and destruction of FEPs. FEPs are also available to Java and OPL applications.

FEP implementations are based on the `CCoeFep` class, which owns a high-priority, invisible control loaded by the Control Environment. Controls are organized as a priority queue. Since FEPs have high priority they receive keyboard events before (nearly all) other controls. The FEP captures and preprocesses sequences of input events which are then returned to the control stack as new events for consumption by lower-priority controls.

Only one FEP instance is allowed per application, since it must run within the application process and thread (in order to access the control stack). A FEP can exist on top of an application without the application being aware of it.

Animation

The animation framework is used to create bitmap-based and sprite-based animations. Animations are created as framework plug-in DLLs (with the extension ANI), which are recognized and loaded directly by the Window Server. While bitmap-based animations are rectangular and restricted to a single window (hence they are also known as 'window' animations), sprites can have irregular shapes and can overlap windows.

Because animations are run inside the Window Server thread, they run with higher priority than would otherwise be possible for any application thread, solving possible problems of slow running due to the high latency of redrawing.

Animations have been used since the early days of the Symbian OS and the framework still contains visible legacy of this, for example in the choice of timing periods.[3]

6.6 A Short History of the UI Architecture[4]

As early as 1997, when the Nokia Communicator project was already underway in Symbian, proposals were made for separating Eikon's look and feel (LAF) from its basic functional machinery. In the end, the Communicator project, like other early licensee projects, settled for adaptation (branching the codeline). However, it was clear that this could only be a short-term solution and that a principled approach was required to support the numbers of licensees and devices which were envisaged.

Reference Designs

As part of the Symbian OS v6 release project, therefore, the earlier look-and-feel separation proposals were revived. The result was Uikon. Its goal was to create a modular, streamlined and extensible user interface framework that would support multiple user interface styles whose look and feel could be customized from a common base. This approach became a central part of the DFRD strategy, which proposed to create reference designs for a generic product matrix that would be licensed to customers as the basis for real products.

Recognizing that each licensee had a distinct product philosophy, the reference designs in effect defined a set of distinct products. Reference designs specified the basic use cases and device style (classic phone or PDA; pen or keyboard input) and physical form factor (tablet or clamshell, as well as screen size, resolution and orientation), and were intended to be followed up with reference implementations including a reference user interface based on custom extensions to Uikon.

Uikon Architecture Evolution

The Uikon architecture consists of a common functional core (Uikon), a standard but non-core supporting library (EikStd), a graphical utility library (EGUL), and a LAF customization framework (UikLaf).

---

[3] See Douglas Feather's Window Server chapter in [Sales 2005].

[4] See also Chapter 16.

Early on, the implementation of common dialogs and controls was split between a core set and an optional set, with printing, file browsing, infrared beaming, and other similar functionality classed as optional.

- Core modules were intended for use unchanged.
- Standard modules were based on the Eikon baseline but were evolved in collaboration with the DFRD teams (Crystal, Quartz and Pearl).
- DFRD-specific libraries were created by DFRD teams.

Initially, 'mixin' classes were used to enable control implementations to reside in LAF-specific custom classes. Invoking `Set()` functions in the mixin classes loaded the custom library dynamically and allowed the core libraries to 'set' the custom concrete implementations.

Largely for performance reasons, this evolved into a stub library model in which the core links statically against a stub library which then loads and initializes the concrete custom library (or libraries, since there may be several). The advantage was that only one copy of the custom DLL was now loaded and one-off initialization was also faster than on-demand initialization. As well as providing a custom library, each variant user interface also implements a LAF module DLL that supplies the specific look-and-feel implementation for the Uikon core, to achieve a consistent look and feel across core, standard and custom libraries. The custom library replaces the Uikon internal library, `UikLafGT`.[5]

In its current architecture, Uikon principally provides application base classes for use by a variant user interface implementation. In early Symbian OS releases, it also provided a core set of controls (such as window borders) and dialogs (standard information and query dialogs).

Additional (optional) standard controls and dialogs, which are directly modified by customizers to form part of a variant user interface, are supplied in the UI Toolkit (part of the larger Symbian Toolkit delivery) and are not described here. Each variant user interface also defines its own custom controls, which vary between user interfaces.

6.7 Component Collections

The UI Framework layer contains two collections of components, as shown in Figure 6.2.



Figure 6.2 Component collections in the UI Framework layer

[5] 'GT' is a legacy Symbian internal name that originally stood for Generic Technology.

UI Application Framework Collection

The UI Application Framework collects the main frameworks related to user interfaces (see Figure 6.3).

It provides generic user-interface framework components that support user-interface customization. Additionally, it provides support directly to applications.

Table 6.1 UI Application Framework Components

Component Name	Development Name
Uikon	`UIKON`
Control Environment (CONE)	`CONE`
FEP Base	`FEPBASE`
UI Look and Feel	`UIKLAFGT`
Uikon Error Resolver Plug-in	`ERRORRESGT`

- The Uikon component provides a concrete framework for user interface and application creation. Applications, typically, should not derive directly from Uikon classes. Instead, they should derive from equivalent classes provided by the variant user interface, because these provide the appropriate look and feel and other device-specific behavior. However, applications implement virtual methods inherited from Uikon and call inherited methods.
- The Control Environment (CONE) provides a control hierarchy and environment. It provides policy-free abstract controls (interactive screen elements) and control context, as the basis for interaction between the user and the application. It includes the application interface to user and keyboard events and View Server encapsulation. Derived concrete controls are provided by the variant user interface. All applications also use CONE (i.e. `CCoeEnv` and `CCoeControl`) directly within the application framework context.

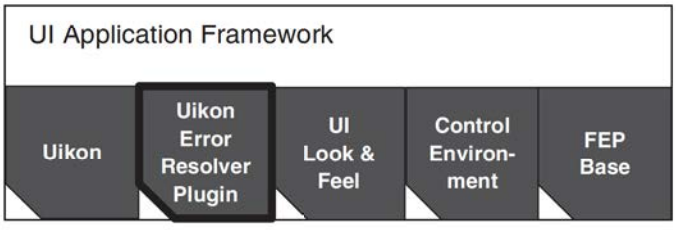


Figure 6.3 UI Application Framework collection

- The FEP Base component provides base classes for creating FEPs. FEPs selectively intercept and preprocess user input events, which are returned to the system as simplified events for handling by applications, to enable keyboard mapping, multitap keyboard input, handwriting recognition, voice recognition and other input preprocessing.
- The UI Look and Feel component defines the look-and-feel properties of the user interface. It defines standard methods (i.e. an API) for which user interface customizers provide an implementation in the UikLaf library of a variant user interface. The role of the user interface LAF component is to provide other parts of the application framework with a way of requesting look-and-feel information from a variant user interface, including the layout and behavior of windows; which bitmaps and fonts to use; and the location of various resource files.
- The Uikon Error Resolver Plug-in is a resource file that maps system-error numbers to helpful error-text strings, which a variant user interface extends and customizes. Errors are flagged when a user interface thread leaves normal execution inside the active scheduler of an application.

UI Support Collection

UI Support (see Figure 6.4) collects miscellaneous frameworks, utilities and libraries that are used by variant user interfaces and which, in some cases, may also be used directly by applications.

- The Graphics Effects component supports flicker-free animation of windows and window contents and composition of animation effects with other graphics objects, to enable GUI special effects (such as animated icons and 'exploding' menus) and moving and resizing windows (known as 'transition effects').
- The UI Graphics Utilities component consists of libraries used by user-interface framework components, the variant user interface and applications. They provide color, font, icon, text, drawing, and number conversion utilities. The utilities include those to query the relative

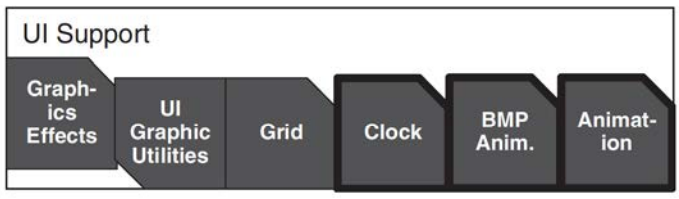


Figure 6.4 UI Support collection

Table 6.2 UI Support Components

Component Name	Development Name
Graphics Effects	GFXTRANSEFFECT
UI Graphics Utilities	EGUL, NUMBERCONVERSION
BMP Animation	BMPANIM
Animation	ANIMATION
Grid	GRID
Clock	CLOCK

positions of nested rectangles, to draw borders, to store color schemes and map logical to physical colors, to perform various font manipulations, to perform number conversions, to find pixel widths of text objects and to package icons as bitmap-plus-mask pairs.

- The Animation component supports window- and sprite-based frame-sequence animation. It enables animated effects to be included in the normal drawing of a window by a client or to be managed server side as a sprite. It also defines a plug-in interface enabling new animation types to be created and loaded as plug-ins directly into the Window Server. Hence, they run in its high-priority thread rather than in an application thread. Sprites can have irregular shapes and can overlap windows. Window animation is used, for example, to create fade effects.
- The BMP Animation component is a Window Server plug-in utility that enables bitmap-based frame-sequence animation. Bitmap-based animations are rectangular.
- The Grid component is a simple layout engine providing presentation, print preview and printing for complete spreadsheets and for spreadsheet cells, rows and columns. It is now considered a legacy component.
- The Clock component is a shared library for creating animation-based digital and analog clocks, used by user interfaces and applications.

7

The Application Services Layer

7.1 Introduction

The Application Services layer provides user-interface-independent support for applications on Symbian OS (see Figure 7.1). Broadly speaking, services whose clients and users are specifically intended to be applications or application engines (rather than system components and servers) can be found here. A number of essential application frameworks are also included. Note that the Java ME implementation also uses the frameworks and services found in the Application Services layer.

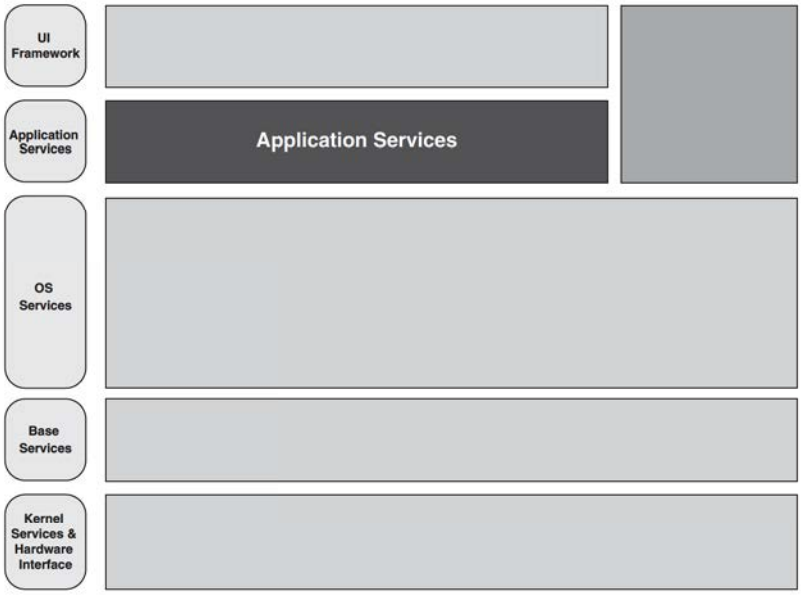


Figure 7.1 The Application Services layer in the system model

Services range from those used by all applications (basic application frameworks), to those providing technology-specific logic (for example, support for device management, messaging and multimedia protocols), to services targeting specific individual applications (PIM and office applications support).

Test or 'reference' user interfaces, where required, are supplied in the customization toolkit for licensees but are replaced in licensee products (including SDKs) and are not described here.

7.2 Purpose

The Application Services layer builds on the underlying services of the operating system to provide services intended primarily for use by applications and their engines, and includes some essential application frameworks which are used by all applications, either directly or as mediated by higher-level frameworks. The Application Services layer is also used by Java ME components.

The Application Services layer provides services used by all applications but mediated by the UI Framework layer and the variant user interface above it, for example, application installation and launching, view switching, and the basic application architecture relationships. It also provides:

- generic services supporting all application types, for example, text rendering and MIME-based content recognition and handling
- technology-specific application support; for example, Versit support (vCard and vCal); alarms for PIM-type applications; and Internet, web and multimedia session protocols
- application-specific services, for example, engines and plug-ins for PIM and office applications; device management; and provisioning.

7.3 Design Goals

From the beginning, Symbian OS has been designed as an application platform. In particular, an important goal has been to make it possible to write rich and compelling applications for pocket-sized, mobile devices (small screen, small ROM and RAM footprint, low power, connected but not 'tethered'). The early system architecture abstracted the application framework as a generic service used by all applications and supplied engines for the built-in application suite independent of the user interface, and layered both beneath the frameworks which supported the GUI-specific aspects of the user interface.

The basic separation of applications into user interfaces and engines and, in particular, the adoption of an MVC-like approach has a long history in Symbian OS. As the system has evolved, there has been an increasing distinction between engines and services. Services are understood as providing generic support for working with data models, for example, generic recognizers, translators and protocol handlers for typed data at the application level. Engines are understood more narrowly as the application-specific logic forming the part of an application implementation that is independent of the user interface. According to this definition, application services would be expected to expose Symbian OS interfaces but application engines would not.

Applying this definition to the system has the effect of moving functionality out of engines (which become narrower in scope and more specific to an application, user interface or vendor), while increasing the common functionality available to the wider set of applications on the phone. This is the direction in which the operating system has been evolving. In the latest operating system releases therefore, the Application Services layer supports application engines but does not include them (except for legacy engines).

There are good reasons for this evolution. Compared with its beginnings, Symbian OS now supports a wider range of devices in diverse markets and geographies. Increasingly, the APIs provided by the generic engines have been perceived by licensees as being too broad (providing too much functionality), while not delivering functionality required in specific markets, for example in Japan and the Far East. Supplying generic engines, with APIs big enough and comprehensive enough to support all application implementations, risks fragmenting the platform rather than unifying it, since licensees are more likely to choose to provide their own specialized (and small) engines than reuse bulky generic engines which nonetheless need extending. Generic engines, in other words, can prove to be a false economy, neither delivering the expected benefits in time to market nor avoiding platform fragmentation.

Providing rich services is a more effective, more generic, more granular, and more customizable way of increasing the capabilities of the platform while serving licensees better.

Some services and application engines may now be considered redundant. For example, Bluetooth profiles are more relevant to phones than WYSIWYG printing; and phones do not typically need a full spreadsheet or word-processor engine, as found in the Office Application engines components.

7.4 Overview

Applications have always been central to the vision of Symbian OS. The original design conception called for more than simply an operating system with an application suite; applications and application support were

considered intrinsic to the operating system. The application architecture was embedded into the object-oriented design and specific application logic – shared data models and data persistence – was provided at the level of operating system services.

- The operating system has evolved to become a common software platform for diverse categories of device, not simply for a single device family as first envisaged.
- The device categories it targets have evolved from PDAs through PDAs with phones to phones, and continue to evolve more generally in the direction of connected, mobile, consumer devices, including phones but not limited to them.
- Open standards have become increasingly important. Efficient and deep integration of open standards for multiple technologies into the operating system platform has become one of its distinguishing features.
- Support for specific, shared data models has become less important, for example the office-style application engines are considered to be legacy functionality.

The Application Services layer includes support for important application-level standards:

- the Versit specification, specifically vCard and vCalendar
- data synchronization, device management and client provisioning, including on-device and 'over the air'
- email standards, including POP, IMAP and SMTP
- phone-messaging standards for GSM and CDMA including SMS, MMS and WAP messaging
- Internet document and data protocols including HTML, XML, WAP, HTTP and Synchronized Multimedia Integration Language (SMIL)
- application-session protocols, Real-time Transport Protocol (RTP) and Session Initiation Protocol (SIP).

Although many of the services based on these standards have been designed to support specific standard applications (messaging and phone applications, for example), they are also generally available to third-party developers creating new applications.

7.5 Legacy Application Engines

The Word, Sheet and Data engines should be considered legacy functionality.[1] While the functionality may continue to feature on specific devices, it should not be considered part of the generic operating system delivery.

Other services, for example printing, should also be considered legacy for different reasons. The original goals of the printing support in Symbian OS (to provide WYSIWYG document printing) have been overtaken by the nature of the content being printed (from photos and contact details to web pages, but rarely a full business document) and by newer protocols (such as the Bluetooth printing profile).

7.6 Architecture

A goal of the user-interface architecture in Symbian OS is to enable as much common functionality as possible on the system side and to make it available to as wide as possible a range of applications. This allows applications to be written with a minimum of new code and the maximum reuse of system-provided code. Applications gain in robustness and reliability because, as far as possible, the most complex code is written only once, on the system side where it is tested and validated, and is reused by application authors. While the strategy for delivering this goal has shifted from providing full application engines to providing comprehensive services, with engines moving up to the licensee layers, the goal remains the same. And while the classes that define the basic architecture of a Symbian OS application differ between variant user interfaces, they all derive from generic Symbian OS classes; Symbian OS implements the underlying generic behavior.

For the application writer, this is interesting. On the one hand, it is extremely powerful, because a little application code goes a long way. On the other hand, having so much richness in the system presents a steep learning curve to the application writer. The Application Services layer provides 'rich system' support for applications.

Application Framework

Model–View–Controller (MVC) is the classic object-oriented abstraction of a graphically based, data-centric, interactive user application. (MVC was originally part of Smalltalk-80 and, according to [Johnson 1998], was 'the first framework that was recognized as a framework'.)

[1] In practical terms, their public APIs are likely to be deprecated in some future release.

Symbian OS, from its first inception, applied an MVC-like model to applications. It is not quite pure MVC, because it elevates the application itself (as an abstraction for system-owned resources) into a first-class concept and because the variant user interfaces do not necessarily code the MVC classes directly. How they interpret MVC is strictly the business of the variant user interfaces.

Applications, documents, UIs and views

The first rule of object orientation (in C++ anyway) is, according to [Koenig and Moo 1997], to 'use classes for concepts'. There are four key concepts in the application model: Application, Document, AppUI and View. The Application Framework supplies the base classes for Application, Document and AppUI, and variant UIs supply appropriate custom specializations. The View class is typically derived directly from `CCoeControl`.

An application is built as a EXE that is recognized by the application architecture and launched in its own process.[2] The framework-defined entry-point function calls the factory function that creates the application instance. The application encapsulates the relationships between the application instance, its document, its document-owned user interface, and its view or views, as well as application-owned resources, for example the application icon and more abstract properties such as UIDs. Applications may have multiple views; every application must have at least one view (i.e. one window-owning or window-controlling control).

Strictly speaking, the application document abstracts a data model and not a file, although applications may be file-based. The document is responsible for storing and restoring the application's persistent data, whether to or from a file or a database. Documents can also be embedded, so that documents may contain other documents (including documents belonging to other applications). The application document is also responsible for creating the application user interface (although the framework takes ownership of the user interface and is responsible for destroying it). Just as the document exists to persist the data state of the application, the user interface exists to manipulate the data state.

A 'data model' in this context really means data plus the APIs defined to create and manipulate it (getters and setters, the 'data logic' defining the translations and other functions that can be applied to the data to return results of some kind). In Symbian OS, this is often loosely referred to as an application 'engine'; the engine is really a code implementation of the machine that transforms the data state, driven by the user interface: the document encapsulates the data model state.

[2] In releases before Symbian OS v9, applications were built as DLL plug-ins and shared process space; the changes are required by the system-wide security model.

'Engine' classes do not have any framework significance (and hence do not derive from a framework class) and they are not required, although they are a useful design pattern for encouraging separation of logic from data.

Since the document creates the user interface and every application needs a user interface, every application must have a document. Each application instance is associated with a single document.

The application view provides a view onto the state of the application data. Views are implemented using controls. On a typical Symbian OS device, desktop user interface idioms (such as multiple overlapping windows) are not appropriate, for a number of reasons: display size is hugely limited compared with a desktop device; handheld operation (and, in particular, one-handed operation) rules out mouse-style interaction, and so on.

Typically, the view metaphor is closer to a stack of sticky notes or a deck of cards. The top card conceals the other cards in the deck. Cards can be brought to the top, shuffled to the bottom of the deck or shuffled unseen within the deck. While applications can have multiple views, only one is visible at any time.

View switching

The View Server provides a framework for sharing application views by 'view switching'. Originally designed to support switching between flip-open and flip-closed modes on the Ericsson R380 (an ER5-based phone), it migrated into the Quartz user interface (which became UIQ) and was eventually adopted back into the operating system. Applications can register views with the server. A registered view owned by one application can then be used by any other application (or indeed by another view in the same application) that requests the view to be activated.

In UIQ, for example, the Contacts application can request activation of the New Message view from the Messaging application when a user taps on an email address in a contact detail. View switching provides a clever shortcut to passing data between applications.

Note that while the View Server manages view switching and owns the framework, it is not used directly by applications: instead, switching is enabled via the application user interface (which is a Control Environment wrapper). View Server uses the Window Server client API to effect view switching.

Support for Generic Applications

While applications are highly dependent on the frameworks supplied by the variant user interface, the underlying support for the application logic is largely provided by Symbian OS. This is an important part of the platform promise that Symbian makes to developers: application logic

should in principle be reusable across the whole range of devices based on Symbian OS. As discussed above, in recent releases the emphasis within the operating system has shifted away from providing reusable application engines towards application services.

Legacy engines

The earliest versions of Symbian OS included a number of fully fledged applications, ranging from standard PIM and Office applications (Agenda, Data, Sheet, Word) to Time World (a time zone browsing and setting application), a Help system, and so on. While there was no Contacts application on the original Series 5, by the time of the later Psion devices (such as the Revo) it had joined the set of standard applications.

Increasingly, providing common services and standardizing APIs is seen as providing more value to licensees than providing ready-made, one-size-fits-all engines. However, the legacy engines still form part of the operating system.

Along with phone-specific functions (messaging and email as well as the phone application itself), PIM applications – most importantly, a phonebook and a simple calendar – are at the heart of what a modern phone provides to its users. Underlying these standard applications are a number of common services, including support for basic text handling, the vCard and vCalendar standards, alarms, backup and restore notifications, and file and date conversions.

Text handling (EText) and formatting (FORM)

Text Handling supports the storing of editable text and its formatting attributes, while Text Formatting provides text view and layout classes (`CTextView`, `CTextLayout`, `MLayDoc`) that control scrolling, selection, cursor management, margin setting, and other attributes of displayed text. Managing display attributes (layout and drawing) is thus distinguished from managing logical text attributes (including text content).

Text content is managed by the text-handling APIs, and consists of Unicode characters, including space characters and paragraph delimiters, as well as formatting attributes, including properties such as paragraph alignment, character fonts, and so on. (Formatting attributes are not the same as text formatting layout attributes.)

The text-handling APIs and the rich text model underlying them have a long history in Symbian OS. They have used Unicode since the ER5u release, the first release to be used in phones, in 1997.

The Text Formatting layout framework is used directly by applications (to lay out text in application user interfaces and documents) and by user interface and system components (to lay out text in dialogs, etc.); for example, text views are used by the Uikon Core API for editable text windows ('editors'), as well as directly by applications to format and display rich text.

vCard and vCalendar

vCard and vCalendar are standards that define formatting conventions for card (address detail) and calendar (diary appointment) entries. The standards allow entries to contain more than simply text (character, number, date and time) content. For example, they can include sounds (for example, alarms) and pictures.

The vCard and vCalendar component provides parsing APIs for vCard and vCalendar entries and enables conversion into Symbian OS native formats.

Alarm server

The Alarm Server manages a queue of system-wide, time-based alarms and provides APIs for applications to set, modify and query alarms. Note that the Alarm Server does not actually notify, sound or show alarms (the Alarm Alert Server performs those functions).

The Alarm Server is a conventional Symbian OS server managing a shared resource (the alarm queue). Clients create a session and connect to the server to use the APIs. The Alarm Server has a long history in Symbian OS.[3]

Backup and restore notification

The backup and restore notification mechanism provides an alert (based on Publish and Subscribe) to signal to PIM applications that backup or restore is in progress or has completed. Applications may need to refrain from writing data to file during backup or may need to re-read files after restore. Other applications should use Publish and Subscribe.

Chinese calendar converter

The Chinese Calendar Converter provides a simple API for converting between Gregorian and Chinese calendar dates.

File converter plug-ins

The File Converter Plug-in is a simple converter that translates HTML to Symbian OS rich text format. It is used, for example, to convert text to HTML email format.

---

[3] Until Symbian OS v7.0s, a single component (known cryptically as EALWL) combined both World Server and Alarm Server functions and served as the engine for the TimeWorld application, see [Tasker 2000, p. 108]. In Symbian OS v7.0s, they were separated and rewritten. The new version of the Alarm Server replaced the old EALWL-specific alarm types (e.g. clock alarms and agenda alarms) to make them more generic.

Printing support

Printing Support implements a framework for managing printers and print jobs, generating graphics input to raster devices and treating printing as a special case of drawing to a device context, much like drawing to a screen or any other display device.

It is intended to be used by applications printing directly to supported printer types and is therefore most suitable for 'old-fashioned' PDA-style applications on 'converged' devices, such as Communicator-style phones, and less appropriate for the more lightweight kinds of application likely to be found, for example, on a phone without a keyboard. For such applications, full WYSIWYG printing is unlikely to be as important as sending a picture to a printer using Bluetooth technology. The print framework can, therefore, be seen as part of the legacy functionality of Symbian OS, along with the Office-style applications it most naturally supports.

It presents a simple application-level interface to underlying printing support provided by the Multimedia and Graphics Services. The printing API, among other things, manages:

- listing and selection of available printers
- encapsulation and setting of the device and print job properties
- selection of a printer port (where required by the printer).

Data synchronization and device management and provisioning

The Open Mobile Alliance (OMA) sponsors data synchronization services based on SyncML, Client Provisioning for OTA device configuration, and Device Management standards. The Application Services layer includes specific support for OMA standards.

Support for Generic Technologies

Standards-based messaging and browsing have become essential functions for mobile phones. The Application Services layer provides extensible support for messaging standards including SMS, MMS and email; for Internet browser protocols; and for newer, session-based multimedia protocols. Supporting services include content recognition, including MIME-type recognition, for data originating from the network; and support for 'smart' messaging (messages containing network-originated configuration and settings data intended to be used by the system rather than read by the end user).

Messaging

Comprehensive support for messaging of all kinds, from email to text and multimedia messages, is an important feature of Symbian OS. Messaging support has been available from the first release. As the operating system has become more phone-centric, messaging has arguably become even more critical than it was originally, although (interestingly) the use cases are subtly different for phones and PDAs.

The Symbian OS messaging implementation provides a complete messaging infrastructure for use by a messaging application, whether from a licensee or other source. It is based around a message server, which manages access to a unified Message Store and performs generic messaging actions that are exposed through a client-side API. It also owns an extensible framework allowing generic actions to be specified for particular message types. The framework is open and is intended to support enterprise-level customization (for example, for bespoke, corporate messaging systems or services) as well as licensee extension and customization (for example, to adapt the generic functionality to a particular user interface idiom – S60 and UIQ messaging applications behave differently from an end-user perspective). The client-side API enables client applications to manipulate the message store, for example, to browse and navigate the message-store folder tree, and provides basic functions, such as edit, copy and move. The framework also supports scheduled sending of messages.

The underlying communications services of the operating system are used to enable message transport over any available network connection, whether phone, short link (Bluetooth or infrared), or cable (serial or USB).

Extensions are provided by Message Type Module (MTM) plug-ins to the framework and the operating system provides product-quality implementations for a standard set of message types, including email (SMTP, POP3 and IMAP4 HTML mail), SMS (on both GSM and WCDMA, that is on 2 and 2.5G, 3G and CDMA 2000 networks: the SMS protocols are specific to each type of network) and MMS.

BIO messaging

An important secondary server and framework is the Bearer-Independent Object (BIO) Messaging Framework, which extends generic messaging to provide a 'smart' messaging server, a message type framework and a watcher framework. Bearer-independence means that the message handling is independent of the type of transport over which the message was received; 'smart' messages are those which are intended for processing by the system, or directly by applications. BIO messaging supports application message types, such as encapsulated vCard and vCalendar data, and system services such as network-access setup messages. The

BIO messaging APIs allow application developers to create their own application-specific 'smart' message types.

The message server provides the underlying mechanisms used by dedicated messaging applications, or other mail or SMS client applications, as well as providing a 'Send As' API as an extension to the client-side API, which allows any application to encapsulate a document and send it as a message type (including Fax), over any available bearer. Any application can also receive messages, using the watcher service, and 'smart' objects.

Messaging support includes handling of MIME and other recognized data types (provided by the Content Handling components); handling of attachments; managing local and remote mail boxes; and editing message contents and properties. The watcher frameworks support alerts for message-related external events, for example a fax-line ringing or an SMS or email being received, and for 'system' messages to be identified and handled.

The basic design principle in the messaging system is to clearly separate generic message handling performed by the framework from the detail of manipulating and handling different message types, which is delegated to the MTM extensions.

The Message Store is considered to be a shared system resource, for which the client–server design ensures multiple simultaneous access by client applications.

The plug-in-framework design allows for a modular and extensible implementation. (However, the MTM model is complex: creating a new MTM is a challenging system-level programming project.) An MTM implements concrete support for three client-side APIs and one server-side API. The client-side implementation consists of a user interface for viewing and editing message contents (and service settings), concrete data, such as icons that clients should display, and the message creation and management functions. The server-side implementation supports manipulation of messages on remote services. Messaging clients link to the client-side MTM. The matching server-side MTM is loaded as needed by the messaging server.

The BIO messaging server and framework is itself implemented as an MTM. BIO messaging plug-ins derive from and implement the framework classes and are loaded by the BIO messaging MTM.

BIO Messaging responsibilities are divided between the MTM (which implements the server and framework), a BIO database (which maps port numbers, MIME types, etc. to BIO types in order to identify the type of incoming BIO messages), and plug-in parsers that parse and process the BIO message payload. Because BIO messages arrive over other message transports, for example as a WAP push or an SMS, watchers are used to receive and tag incoming BIO messages. Watchers that watch for specific message types are created by deriving from and implementing the watcher framework classes.

The scheduled send framework is implemented by the Server MTM and provides classes that define the scheduling parameters, allowing messages to be scheduled (sent later), rescheduled or deleted from the schedule. MTM implementations for different message types can choose whether or not to support message scheduling.

At Symbian OS v9, the supported message types include email (POP3, IMAP4 and SMTP), SMS and OBEX. MMS messaging, which was included in Symbian OS v8, may be provided as part of a licensee user interface implementation.

Content handling

An important aspect of supporting messaging, browsing, and other network-oriented applications is the provision of content recognition, parsing and access services for protected content (key, certificate or other DRM-protected downloads, for example).

Symbian OS provides standard application services that support:

- file and data recognition based on MIME types (MIME Recognition Framework), standard web types (Web Recognizers) and multimedia file types (MMF Recognizers)
- parsers and handlers to support SMIL (SMIL Parser) and 'smart' messages and content (BIO Messaging Parsers) and WAP 'push' messages (WAP Push Handlers)
- handling and providing access to DRM-protected content (Content Access Framework for DRM).

These services are used by applications either indirectly via the various application-level messaging, web and multimedia frameworks and services or directly through the Application Architecture recognizer interface. These services are also used by system components, for example, the messaging framework.

The Application Architecture provides a 'Recognize Data' interface which is implemented by plug-ins to the MIME Recognition Framework. This enables recognition of non-native document types in order to associate documents with applications. (Native document types are identified and associated with applications using UIDs). Associating documents with applications allows appropriate applications to be started (or offered to users) when a user performs an action to open a document, as well as allowing default documents to be located when applications are launched. Data types as well as documents can be recognized.

File and data recognizers are written as plug-ins to the MIME Recognizer Framework (from Symbian OS v9 they conform to the ECOM Plug-in Framework) and are scanned for and loaded during operating system startup.

Data recognizers are provided for common MIME types, URLs, web bookmarks, HTML and XML, and multimedia file types. The supported multimedia types depend on the licensee implementation of multimedia plug-ins for supported media types.

Applications can register with the Application Architecture as handlers for specified MIME/data types. The Application Architecture maintains a list of all recognizers in the system and their supported data types.

The WAP Push handlers are intended to support WAP browser applications. They are plug-ins to the WAP Push Framework and respond to WAP Service Initiation (SI) and Service Load (SL) signals to take ownership of incoming messages and validate, parse, and extract the message content. SI and SL messages signal actions to WAP browser applications (to display content or a URL), unlike other WAP Push message types (MMS and OTA), which are pure message carriers for messaging, not browsing, services. The Web Push handlers are intended to support WAP browser applications directly.

The Content Access Framework provides a generic mechanism to support DRM implementations, based on defined interfaces for brokering controlled content between content agents (DRM applications) and content-consuming applications (for example, media players).

The BIO Messaging parsers plug into the BIO Messaging Framework to enable parsing of specific BIO message types, including vCard business cards, email notifications, Nokia Smart Messages and Nokia and Ericsson OTA setup messages. (Note that BIO Messages use WAP messaging.)

The SMIL Parser is an XML parser that uses a 'mini-DOM'-like API to parse and validate XML against simple DTDs. SMIL is an XML language that defines presentation attributes for encoded text, images, video and audio. It is provided primarily to support handling of MMS messages with SMIL content. (Note the earlier remarks about MMS not necessarily being supported on all devices from Symbian OS v9.) The parser however is also available for direct use by applications and provides APIs to perform simple XML parsing (not limited to SMIL). Heavier duty, generic XML parsing is provided by Base Services components.

A SMIL parser was first introduced in Symbian OS v7.0. The current implementation, which is able to parse any XML document against a simple DTD, was introduced in Symbian OS v7.0s and the original parser was deprecated.

Internet, web and multimedia protocol support

A number of components provide infrastructure support for Internet and web applications including web and WAP browsers and WAP messaging. Newer protocols such as RTP and SIP have also been introduced in the latest releases of the operating system to support new categories of interactive streaming applications.

Basic Internet, web and WAP support consists of framework, utility, and application engine components providing application-level interfaces to Internet protocols (HTTP, Telnet) and WAP Push messaging. The HTTP Transport Framework provides a generalized client interface for applications wanting HTTP transport sessions over TCP/IP or WSP sessions (the WAP equivalent of HTTP).

The HTTP Transport Framework provides a complete supporting framework for HTTP and WSP applications, such as HTML or WAP browsers. For WAP browsing the underlying support of a full WAP stack is required; this is no longer part of the Symbian OS delivery and therefore depends on the licensee platform to provide a full WAP stack. The framework adopts a session model based on a core client API and request–response message exchange transactions with a remote URL.

The WAP Push components provide an interface between the WAP stack and the messaging infrastructure to support WAP as a messaging transport. The WAP Push components are used by the messaging services, to support receiving WAP push messages and BIO messages, and by other system components including, for example, Java. Note that simple client access to WAP push is provided by the WAP Message API of the WAP Stack.

Implementers of a WAP stack need to be aware of the dependency of the HTTP Transport Framework on it. In effect, the lower level of the framework serves as an adaptation layer to the WAP stack, implying that work is required to adapt it to a WAP stack implementation.

The Telnet and FTP engines are rather simple application-level services based on clients creating a client session to the Symbian Telnet or FTP daemon, through which the client can conduct a dialog with a specified host. (Note that FTP does not expose public APIs.)

More specialized Web browsing support is provided by the stand-alone Bookmark Support component, which provides access to a bookmark database and APIs for creating, reading and deleting bookmarks and creating a folder tree. The database uses the Central Repository to store all data. There is only one bookmark database.

A folder object contains an array of `CBookmarkBase` objects. A bookmark must contain a URI, authentication data, the last time it was visited and an indication if it is the home page. Applications can set item visibility to public or private.

HTTP transport framework

The HTTP Transport Framework is based around a Core API, which manages the client-session interface to a session based on either a WSP or HTTP protocol, for example for WAP or web browsing. In both cases, secure versions of the protocols are also supported. Within a session, message-based transactions are conducted with the remote

URL. A message is a generic abstraction that packages contents of any type.

As well as the Core API, clients can configure a session to use Filter Plug-ins that are loaded by the framework and used by applications to handle, process or modify message content. Default filters are provided for message authentication, redirection and validation.

Beneath the Core API, protocol handlers and transport handlers interface to the underlying transport. WSP and HTTP protocol handlers are supplied by default. WAP Stack, WAP WTLS, HTTP and HTTPS transports are available. WAP and WTLS use the WAP Stack interface directly; HTTP and HTTPS use the Socket Server to provide TCP/IP sockets or secure sockets, in all cases using an appropriate network interface.

Real-time transport protocol

The real-time transport protocol (RTP) is a network transport service that provides real-time guarantees on packet latency to support uses such as interactive audio and video, for example, web conferencing. TCP-based packet services have a (relatively) high potential latency. For many applications, heuristics (buffering, selective dropping and repeating of packets, etc.) can be used to maintain service quality at a satisfactory level, even for demanding applications such as streaming. However, two-way interactive services have effective real-time requirements which cannot be met simply by smoothing packet arrival latencies.

RTP implements reliable and real-time bound transport using UDP packets over IP. RTP services support payload-type identification, sequence numbering, time stamping, and delivery monitoring of packets.

From a system perspective, RTP is provided to support the Multimedia Framework introduced in Symbian OS v8. It is designed as a core software stack that implements RTP/RTPC packet creation and handling using the underlying network infrastructure, and an upper API used by applications, which link to it.

RTP is available to applications using a socket interface. From the user perspective, it is created and used in essentially the same way as any socket-based transport. Within a socket server session, an RTP subsession is opened.

RTP provides APIs to:

- create and manage RTP sessions
- register for and handle events
- manage and access RTP packets and reports
- create, send and receive packet streams
- manage, send and receive reports.

RTP was introduced in Symbian OS v9.

Session initiation protocol

Session Initiation Protocol (SIP) is a simple but powerful protocol enabling peer-to-peer, multiple-participant sessions to be created over a TCP/IP packet network. The protocol is reminiscent of HTTP (in its use of URLs to identify participants) and SMTP (plain text messages). SIP messages are used to set up and terminate sessions.

The SIP Framework integrates a plug-in implementation into the underlying network infrastructure, including RTP. The operating system provides only the framework; licensees supply the service implementation.

The SIP Framework was introduced in Symbian OS v9.2.

Other Application Services

Improved secure installation services provided by the App Installer and a System Starter that manages server startup at boot time were introduced in Symbian OS v9.1 to improve the supporting infrastructure for applications (although they do not expose APIs directly to applications).

The App Installer uses the Certificate and Key Management services (see Chapter 8) provided by lower layers of the system to manage certificate- and key-secured applications.

The System Starter, while it does not expose public APIs, is configurable by licensees. In the original design of Symbian OS, true reboots were assumed to be rare events. The operating system was designed to support devices that would run for months and even years at a time between reboots. Booting-up time was therefore an insignificant cost. However, the phone use case is very different. Phone users switch phones off frequently and expect a fast boot when they switch them on.

Symbian's server model – ubiquitous use of servers to manage all system resources, logical and physical – leads to multiple servers being started at device boot time with a cascade effect. (Any server can arbitrarily start many other servers; in the past, some have.)

The System Starter allows a start-up policy to be specified and enforced. This enables careful management of the start-up sequence, to enable a device to become maximally responsive in minimum time, even if loading of the full server set continues in the background. If a server run list is found, it is used to select which servers start and in which order. In this scenario, some servers are not started until they are first called by a client.

7.7 Component Collections

The Application Services layer contains the component collections shown in Figure 7.2.

- System level services:
  - Application Framework

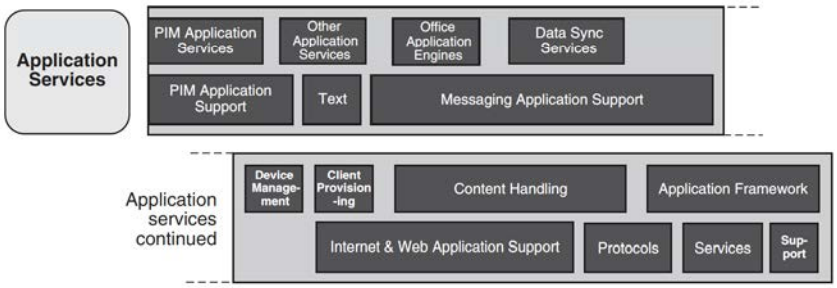


Figure 7.2 Component collections in the Application Services layer

- Application Launch Services
- Multimedia Protocols

- Application services and engines:
  - Data Sync Services
  - Device Management
  - Client Provisioning
  - PIM App Services
  - Other Application Services
  - Office Application Engines

- Lower-level application support:
  - PIM Application Support
  - Messaging Application Support
  - Content Handling
  - Internet and Web Application Support
  - Printing Support

Application Framework Collection

- The Application Architecture component defines the key application responsibilities and interactions with data and the user interface.

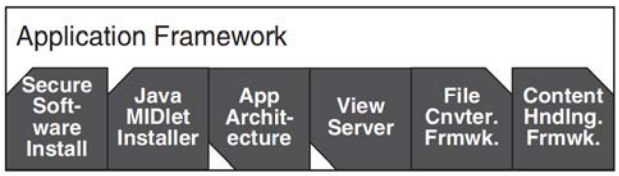


Figure 7.3 Application Framework components

Table 7.1 Application Framework Components

Component Name	Development Name
Application Architecture	APPARC
View Server	VIEWSRV
File Converter Framework	CONARC
Content Handling Framework	CONTENT_HANDLING
Secure Software Install	SECURESOFTWAREINSTALL
Java MIDlet Installer	JAVAMIDLETINSTALLER

It encapsulates the key application classes, which are abstracted via Uikon and, ultimately, by a vendor-specific variant user interface.

- The View Server component provides a mechanism for view sharing and view switching between applications. A running application can switch into and use a view belonging to another application.
- The File Converter Framework component supports creation of file converter plug-ins that enable applications to request file-to-file conversion based on the MIME types of the files. It is typically used to support conversion between Microsoft Office and Symbian OS proprietary formats.
- The Content Handling Framework component contains the File Converter and Content Handling frameworks, which are used to provide applications with common framework behavior independent of the user interface. The Content Handling Framework supports the finding, loading, processing and displaying of typed content by content handlers on behalf of applications.
- The Secure Software Install and Java MIDlet Installer components enable the installation of native applications and Java MIDlets. SIS files, based on versions of Symbian OS that do not include platform security, do not install on devices based on Symbian OS v9.

Application Launch Services Collection

This collection (see Figure 7.4) contains only one component, which enables policy-based startup of system servers at boot time. The server startup sequence is defined in a policy file, which can be customized by

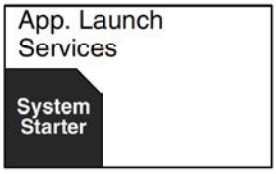


Figure 7.4 Application Launch Services components

licensees to tune boot-up time and ensure that the device is responsive to the user as quickly as possible after switch on.

Table 7.2 Application Launch Services Components

Component Name	Development Name
System Starter	SYSSTART

Multimedia Protocols Collection

This collection (see Figure 7.5) provides support for the real-time transport protocol (RTP) and the session initiation protocol (SIP).

Table 7.3 Multimedia Protocols Components

Component Name	Development Name
RTP	RTP
SIP Framework	SIP_COM
SIP Connection Provider Plug-ins	SIPCPR, SIPDUMMYPRT, SIPSTATEMAC, SIPPARAMS, SIPSCPR

- The RTP component is a server- and user-side API providing socket-based access to RTP services. It provides an IP-based real-time network transport service.
- The SIP Framework and SIP Connection Provider Plug-ins provide support for SIP and integration into the networking infrastructure. It

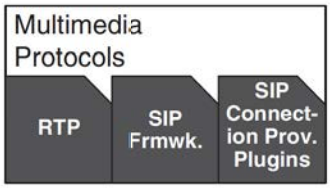


Figure 7.5 Multimedia Protocols components

does not provide the protocol implementation (which is provided as a plug-in by licensees). SIP is the main signaling protocol for 3GPP and is used by phone, multimedia and messaging applications.

Data Sync Services Collection

This collection (see Figure 7.6) provides support for data synchronization.

Table 7.4 Data Sync Services Components

Component Name	Development Name
Sync Initiation	SYNCMLINITSERVER
OMA SyncML Framework	SYNCMLCLIENT
OMA SyncML DM Interface	SYNCMLDMCLIENT
OMA Data Sync	SYNCMLDSCLIENT

SyncML is an open industry standard, primarily for data synchronization but extending to device management. SyncML has been adopted and standardized by the Open Mobile Alliance.

The SyncML protocol supports data providers (for example, application engines) and components requiring remote management (for configuring settings, for example) over various transports. It is implemented as a server, with a supporting plug-in framework, that supports synchronization and device management over HTTP, WSP and OBEX.

Client Provisioning Collection

This collection (see Figure 7.7) provides components for client provisioning.

Client Provisioning is an OMA standard for configuring application and network settings on mobile devices. It overlaps to some extent

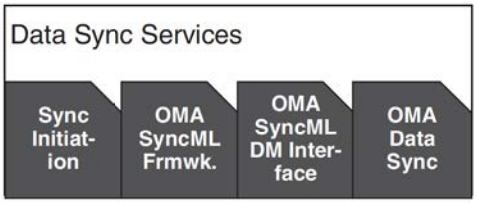


Figure 7.6 Data Sync Services components

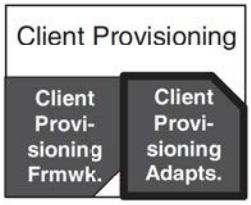


Figure 7.7 Client Provisioning components

Table 7.5 Client Provisioning Components

Component Name	Development Name
Client Provisioning Framework	`DEVPROV_CLIENTPROV_FRAMEWORK`
Client Provisioning Adaptors	`DEVPROV_CLIENTPROV_ADAPTERS`

with SyncML and competes with proprietary alternatives (Nokia Smart Messaging and Nokia and Ericsson OTA).

Client Provisioning supports components requiring either one-off settings configuration (network settings setup, for example) or continuous provisioning (for example, settings management plug-ins to applications or Symbian OS services) over various transports.

Device Management Collection

This collection (see Figure 7.8) provides a framework and plug-ins implementing OMA Device Management based on SyncML and supporting Remote Terminal Management and continued provisioning of devices by network operators.

PIM Application Services Collection

This collection (see Figure 7.9) provides specialized support specifically for the Agenda and Contacts applications.

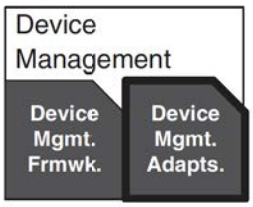


Figure 7.8 Device Management components

Table 7.6 Device Management Components

Component Name	Development Name
Device Management Framework	DEVPROV_DEVMAN_FRAMEWORK
Device Management Adaptors	DEVPROV_DEVMAN_ADAPTERS

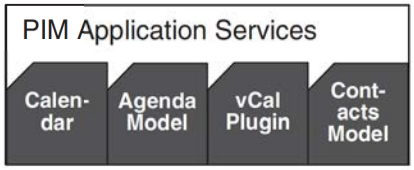


Figure 7.9 PIM Application Services components

Table 7.7 PIM Application Services Components

Component Name	Development Name
Contacts Model	CNTMODEL
Calendar	CALINTERIMAPI
Agenda Model	AGNMODEL
vCal Plug-in	AGNVERSIT

- The Contacts Model component is an application model providing a common contact or address book API implemented over an underlying database.
- The Calendar component is intended to replace the previous Agenda Model API. Calendar provides a cut-down API more suitable for a modern phone. The Agenda Model API is larger and has its origins in the needs of PDA users. Calendar partially supports the iCalendar standard. The vCal Plug-in is a library used by the Agenda Model to communicate with the vCard and vCal components.

Other Application Services Collection

This collection (see Figure 7.10) provides miscellaneous application support, originating from the Series 5 set of built-in applications, but extended more recently with the addition of the Timezone component.

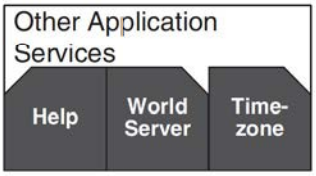


Figure 7.10 Other Application Services components

Table 7.8 Other Application Services Components

Component Name	Development Name
Timezone	`TZ, TIMEZONELOCALIZATION, TZLOCALIZATIONRSCFACTORY, TZCOMPILER, TZDB`
World Server	`WORLDSERVER`
Help	`HLPMODEL`

- The Timezone component provides localization support, including a time-zone database, for Standard, Daylight, Short Standard and Short Daylight names for time zones. Localized names are stored in the resource file framework. Users can create cities and link them with time-zone information. Cities can also be grouped irrespective of time zone.
- The World Server component originated in the Time/World application of the original EPOC release. It is based on a world cities' database and server, and allows setting and easy switching between 'home' and 'away' locations and time zones, as well as time-zone browsing. It was deprecated in Symbian OS v8.1, in favor of the Timezone component.
- The Help component provides an engine implementation of a context-sensitive help system, providing read-only access to all help files on a Symbian OS device. Help files are essentially heavily compressed databases, each containing a series of topics relating to different applications or subjects.

Office Application Engines Collection

This collection (see Figure 7.11) provides legacy application-engine implementations of the original EPOC built-in applications: Data (database), Sheet (spreadsheet), and Word (word processor). Redundant on a modern phone, they are likely to be removed in a future operating system release.

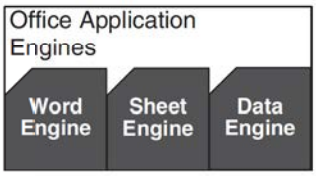


Figure 7.11 Office Application Engines components

Table 7.9 Office Application Engines Components

Component Name	Development Name
Data Engine	DAMODEL
Sheet Engine	SHENG
Word Engine	WPENG

PIM Application Support Collection

This collection (see Figure 7.12) provides services that may be useful to a variety of applications and application engines but which, typically, are quite closely tied to legacy applications.

Table 7.10 PIM Application Support Components

Component Name	Development Name
Alarm Server	ALARMSERVER
vCard and vCal	VERSIT
Chinese Calendar Converter	CALCON
File Converter Plug-ins	CHTMLTOCRTCONVERTER, CONVERT, RICHTEXTTOHTMLCONV
Backup Restore Notification	BACKUPRESTORENOTIFICATION

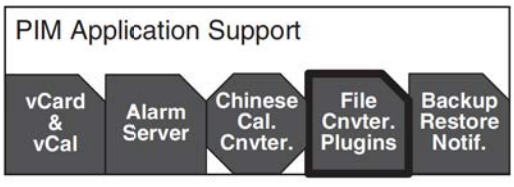


Figure 7.12 PIM Application Support components

- The Alarm Server component manages a queue of system-wide, time-based alarms, providing set, modify, query and notify APIs for client applications.
- The vCard and vCalendar components are parsers that convert between vCard or vCalendar entries and Symbian OS native formats.
- The Chinese Calendar Converter component provides a simple API for converting between Gregorian and Chinese calendar dates.
- The File Converter Plug-ins component supports conversions between HTML files and Symbian OS rich text objects stored in files, and between specific formats, for example Microsoft Excel, Microsoft Word and Microsoft font formats, and Symbian OS native rich text.
- The Backup Restore Notification component is used by legacy applications to notify of system-wide backup and restore operations. Publish and Subscribe provides a preferred alternative for new applications.

Messaging Application Support Collection

This collection (see Figure 7.13) provides Messaging and BIO Messaging frameworks and MTM plug-ins.

- The Message Store component provides a message server and framework, supporting standard message types (for example email and SMS).
- The BIO Messaging Framework component supports 'smart' message types (Bearer-Independent Objects), for example vCard or vCalendar messages and network setup messages.
- The BIO Watchers component provides a framework and service for notification of message arrival to applications.
- The Scheduled Send MTM component supports scheduled sending of any available message type and defines the scheduling parameters.
- The Email MTM components are plug-ins to the Message Store framework providing support for sending, receiving or editing POP3, IMAP4 (HTML mail) and SMTP email messages.
- The OBEX MTM components are plug-ins to the Message Store framework providing support for OBEX messages.

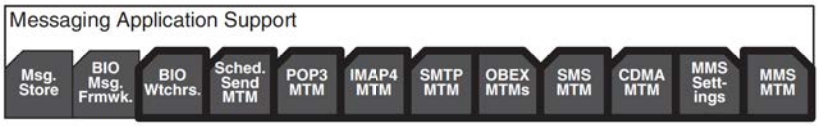


Figure 7.13 Messaging Application Support components

Table 7.11 Messaging Application Support Components

Component Name	Development Name
Message Store	MSG_FRAMEWORK
BIO Messaging Framework	MSG_BIOMSG
BIO Watchers	MSG_BIOWATCHERSCDMA
Scheduled Send MTM	MSG_SCHEDULEDSEND
POP3 MTM	MSG_EMAIL
IMAP4 MTM	IMAPSERVERMTM
SMTP MTM	SMTPSERVERMTM
OBEX MTMs	MSG_OBEXMTM
SMS MTM	MSG_SMS8.1
CDMA MTM	CDMASMSMTM
MMS Settings	MSG_MMS_SETTINGS
MMS MTM	MMS

- The SMS and MMS MTM components are plug-ins to the Message Store framework providing SMS message support for GSM/WCDMA and CDMA 2000 and the infrastructure support for MMS messages. From Symbian OS v9, licensees may provide the MMS MTM.

Content Handling Collection

This collection (see Figure 7.14) provides frameworks, handlers, parsers and recognizers for typed data and documents (including MIME and web types, SMIL and BIO messages) and DRM content.

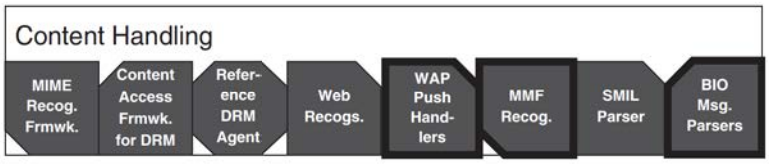


Figure 7.14 Content Handling components

Table 7.12 Content Handling Components

Component Name	Development Name
SMIL Parser	`GMXML`
MIME Recognizer Framework	`EMIME`
WAP Push Handlers	`WAPPUSHSUPPORT`
Web Recognizers	`RECOGNIZERS`
Content Access Framework for DRM	`CAF2 , CAF2CONFIG`
Reference DRM Agent	`DRMAGENT`
MMF Recognizers	`RECMMF`
BIO Messaging Parsers	`CBCP, ENP, GFP, IACP, WAPP`

- The SMIL Parser component parses SMIL content based on a generic XML Parser and Composer with a 'mini-DOM' API able to perform syntax checking against simple DTDs. It replaces the SMIL Translator implementation of Symbian OS v7.0s.

- The MIME Recognizer Framework component supports for MIME data types.

- The WAP Push Handlers components are plug-ins to the WAP Push Framework implementing handlers including Several Interfaces, Single Logic (SISL).

- The Web Recognizers component supports URLs and web bookmarks and are implemented as plug-ins to the MIME Recognizer Framework.

- The Content Access Framework for DRM component provides generic APIs for brokering DRM-protected content between agents (DRM applications) and consumers (e.g. media players). It includes a reference DRM-agent implementation.

- The MMF Recognizers component provides support for multimedia data and document types.

- The BIO Messaging Parser components parse by BIO message type.

Text Rendering Collection

This collection (see Figure 7.15) enables not just applications but any components that want to display or manipulate text to use the Symbian OS text-handling and formatting APIs.

Table 7.13 Text Rendering Components

Component Name	Development Name
Text Formatting	FORM
Text Handling	ETEXT

- The Text Formatting component provides text view and layout classes to control scrolling, selection, cursor management, margin setting, and other attributes of displayed text. It supports the separation of display attributes (layout and drawing) from logical text attributes (styles). It is used, for example, by the Uikon Core API for editable text windows and, more generally, by applications to format rich text.
- The Text Handling component supports the storage of editable text and its formatting attributes, for example, paragraph alignment and character fonts. It is used with the Text Formatting text view APIs.

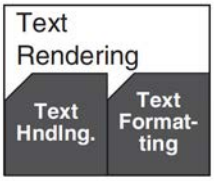


Figure 7.15 Text Rendering components

Internet and Web Application Support Collection

This collection (see Figure 7.16) provides Internet, web and WAP application support.

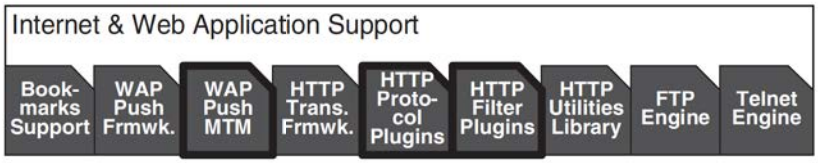


Figure 7.16 Internet and Web Application Support components

Table 7.14 Internet and Web Application Support Components

Component Name	Development Name
HTTP Transport Framework	HTTP
HTTP Protocol Plug-ins	HTTP
HTTP Filter Plug-ins	HTTP
HTTP Utilities Library	INETPROTUTIL
Bookmark Support	BOOKMARK_SUPPORT
Telnet Engine	TELNET_E
FTP Engine	FTP
WAP Push Framework	WAPPUSH
WAP Push MTM	WAP-BROWSER

- The HTTP Transport Framework component enables clients to establish a transport session for HTTP-like protocols, provides core APIs for transport sessions, transactions, and messages.
- The HTTP Protocol Plug-ins component provides dynamically loaded application and network protocol handlers, including TCP/IP, HTTP 1.1 and WSP 1.2.
- The HTTP Filter Plug-ins component provides dynamically loaded plug-ins to configure a transport session before use. It includes default HTTP and WSP filters that encapsulate responses to session events, for example, client authentication, message validation and message redirection.
- The HTTP Utilities Library component stores utility classes commonly used by Internet protocol parsing components. It contains implementations for URIs, a standardized time format, and simple text parsing utilities.
- The Bookmark Support component provides a bookmark database for web browsers.
- The Telnet Engine component provides a Symbian OS Telnet daemon and supports client sessions for communicating with a specified host.
- The FTP Engine Symbian OS FTP daemon, supports client sessions for communicating with a specified host. Does not expose public APIs.

- The WAP Push Framework component provides an interface between the WAP stack and the messaging infrastructure to support WAP as a messaging transport.
- The WAP Push MTM component provides a WAP stack implementation supporting messaging interfaces.

Printing Support Collection

This collection (see Figure 7.17) provides standard dialogs for setting up print jobs and controlling access by application clients. It is considered a legacy component for most devices.



Figure 7.17 Printing Support components

Table 7.15 Printing Support Components

Component Name	Development Name
Printing Services	PRINT

8

The OS Services Layer

8.1 Introduction

The OS Services layer (see Figure 8.1) provides the servers, frameworks and libraries that implement the core operating system support for graphics, communications, connectivity, and multimedia, as well as some generic system frameworks and libraries (Certificate and Key Management; the C Standard Library) and other system-level utilities (logging services). In effect, it is the layer that extends the minimal base layers of the system (the kernel and the low-level system libraries that implement

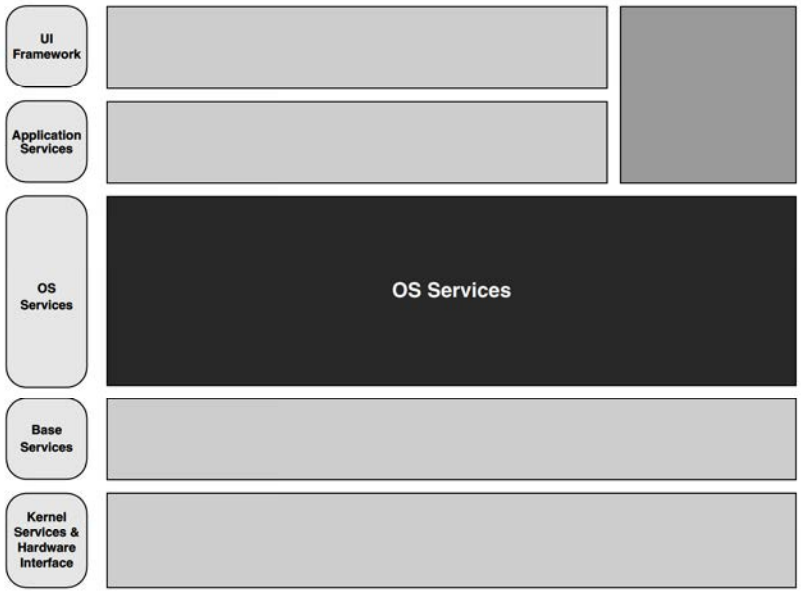


Figure 8.1 OS Services layer in the system model

the basic OS primitives and idioms) into an extensible, programmable, and useful operating system.

In terms of the number of components, it is by some margin the largest single layer of the system. To bring clear structure to it, the System Model organizes the layer into four major blocks by broad technology type (see Figure 8.2):

- Generic OS Services
- Comms Services
- Multimedia and Graphics Services
- Connectivity Services

These blocks are relatively self-contained. (Generic OS Services is used by the other blocks in the layer; Connectivity Services uses the transport technologies of Comms Services.)

This chapter describes the Generic, Multimedia and Graphics, and Connectivity Services blocks; the Comms Services block is described in Chapter 9.



Figure 8.2 Blocks of OS Services components

8.2 Purpose

Symbian OS is a microkernel operating system. The kernel is restricted to providing the minimum of essential services, specifically those required to implement process execution and memory access models. These are extended by the remaining (non-kernel) components of the base layers of the system, to support bringing up a bare system on hardware, providing access to peripherals and a file system, and to support a program execution model. Higher-level system services are built on top of this foundation.

In Symbian OS, the higher-level system services are located in the OS Services layer. These services provide the specialized system-level support required by other system components and by higher layers of the system, as well as by applications. Thus, for example, graphics support, communications support including networking and telephony, and the connectivity infrastructure are all provided as OS services.

Generic OS Services Block

This block provides a small number of generic services for use directly by applications, as well as some specific programming libraries intended for application and system use (including for use by the user interface and application support layers above).

- The logging and task-scheduling services are used by applications as well as by system components.
- The C Standard Library, providing a basic POSIX environment, is used by system components (for example, Java) and is also useful to those porting software from other platforms.
- There are libraries and frameworks supporting cryptographic and certificate-based security, including the key and certificate stores.

Multimedia and Graphics Services Block

This block provides all graphics services above the level of hardware drivers and provides the frameworks supporting multimedia services.

- It provides windowing, event handling, bitmap and vector graphics support including all font, drawing and bitmap functions, as well as low-level support for WYSIWYG printing.
- It defines a comprehensive set of multimedia APIs and provides a framework for implementation. It includes camera and broadcast tuner APIs, sound capture and recording APIs, still and moving image capture and recording APIs, display and play APIs, and conversion and manipulation APIs.



Figure 8.3 Generic OS Services block

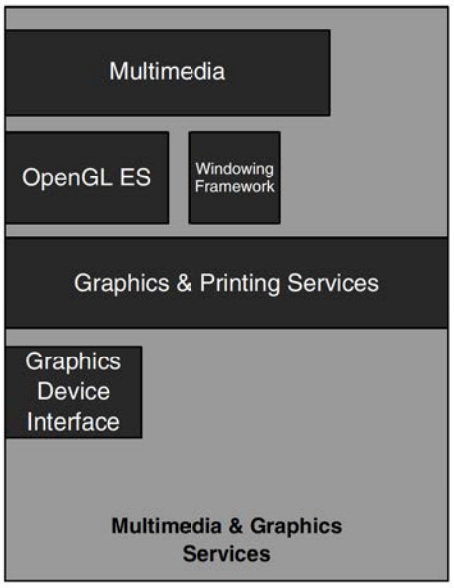


Figure 8.4 Multimedia and Graphics Services block

Connectivity Services Block

This block provides the device-side support for connectivity services, for example backup and restore, file transfer and browsing and application installation. (Data synchronization is provided in the Application Services layer, see Chapter 7.)

8.3 Design Goals

While the detail has changed considerably, most of the services located in the OS Services layer can be traced back to the original, early architecture of Symbian OS. In the earliest designs, the principal communications transport technology was serial, although networking support and, in particular, thorough support for standard Internet protocols had already been identified as an essential requirement, leading to the design of a networking infrastructure tightly bound to the communications services.

The first work on telephony-specific services, meanwhile, was well underway even before the first release of the OS, and relevant requirements were being evolved in collaboration with licensees. While at that time there were no specific multimedia services, the bitmap-based, windowing graphics system was central, and support for various audio formats was present from the beginning.

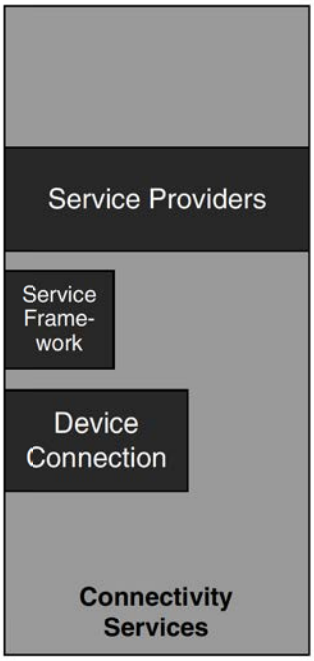


Figure 8.5 Connectivity Services block

Connectivity was also considered a vital service from the beginning, although it was a significantly simpler service based on Symbian's proprietary PLP protocol, a simple data transfer protocol over a physical (wired) serial port or emulated serial port over IrDA.

Since then, the rapid evolution of mobile telephony through successive technology generations, the ubiquity of the Internet and the increasing packetization of services, and the emergence of data exchange standards and protocols such as SyncML have all been powerful forces in shaping the evolution of the OS Services. The rapid convergence of multiple device functions with mobile phones has also had a dramatic impact on the kinds of services required from Symbian OS. Above all, multimedia technologies, which a decade ago were the province of top-end workstations, have migrated inexorably downwards onto smaller devices; simultaneously new categories of multimedia device have been invented (digital music players and digital cameras). New technologies, including digital broadcast TV for mobile devices and session-based multimedia protocols enabling two-way, real-time video and audio applications, continue to emerge and evolve.

In Symbian OS, providing support for all such services falls squarely in the realm of the OS Services layer.

8.4 Overview

All the core system servers, with the exception of the kernel server and the file server, are found in the OS Services layer.

- Generic OS Services:
  - The Task Scheduler provides a task-launching service for time-based and condition-based task triggers.
- Multimedia and Graphics Services:
  - The Window Server provides access to screen hardware and application and system events.
  - The Font and Bitmap Server provides font and drawing contexts for all bitmap-based devices.
- Connectivity Services:
  - The Software Install Server provides a secure software installation interface from remote clients.[1]
  - The Remote File Server provides a file system interface from remote clients.
  - The Secure Backup Socket Server provides a backup and restore interface from remote clients.

In addition, the essential communications servers appear in this layer (see Chapter 9):

- Comms Framework and Serial Comms
- Telephony
- Networking

Together these servers provide interfaces to system-level support for almost all the major services provided by the OS above the level of the kernel (persistent store and file system services are the notable exceptions). OS Services therefore really can be thought of as the essential infrastructure on top of which all application-level services are built.

It is also the location of Symbian OS support for many open standards including:

- OpenGL ES, FreeType, and graphics and audio file formats including GIF, BMP, WAV, MP3

[1] The Remote Software Install Server is *not* the same as the Secure Software Install Server; the former is used only by Connectivity Services components and manages software installation from a connected host device, typically a PC; the latter is the trusted computing base gatekeeper installation component, see Chapter 7.

- Cryptographic and key standards including RSA, DSA, DH, DES (not for use by end-users)
- ANSI C Standard Library, POSIX
- TCP/IP v4 and v6 networking
- 2G, 2.5G, and 3G telephony for GSM/UMTS and CDMA2000
- Serial RS232, USB, Bluetooth, Infrared/IrDA, OBEX
- Fax

From an application perspective, many of the provided services are sufficiently specialized that few applications use them directly, or even at all. Their functions are exposed to applications through higher-level frameworks and, for example, only specialized applications explicitly use telephony or networking, although any application may use the SendAs API.

However, all applications use the font services and perform event handling, window management, and drawing. Whether they know it or not, all applications depend on these core servers.

From another perspective, this layer brings out many aspects of the particular character of Symbian OS: multiple essential services are provided independently of the kernel; the client–server and framework and plug-in design patterns are ubiquitous; and object-oriented design idioms are widespread.

8.5 Architecture

The system model captures the broad division of responsibilities between components in the block structure of the layer. In general, each block is structured around one or more servers that collaborate to deliver a set of related services. Typically, servers also provide a plug-in framework, enabling extensible and flexible implementation of the underlying services. Frequently, the design includes multiple levels of frameworks through which services are implemented.

As an approximation, the key interfaces to each block are encapsulated in the principal servers and frameworks each block contains, although in many cases there are also additional utilities exposing library interfaces. In general, each block can also be thought of as layered to form a logical stack. The topmost layers of the stack expose the client interfaces (used by applications and system clients); the middle layers typically interface to other system services; and the lower layers expose framework interfaces (used by device implementers to create hardware adaptation plug-ins).

8.6 Generic OS Services Block

The Generic OS Services block provides a number of general-purpose utility-style services which are useful to applications (and other system



Figure 8.6 Generic OS Services block

components) and some specific frameworks and libraries that provide useful system services.

The frameworks and libraries include an implementation of the C Standard Library and framework support for secure certificates, keys and tokens. The more general-purpose utilities include logging and scheduling services and some legacy components.

Design Goals

For the most part, the Generic OS Services are system utilities or libraries which have (or had in the past) some specific association with particular applications but which have been seen as more generally useful for both system and application support and so have migrated downwards in the system as their services have been generalized.

The Event Logger and Task Manager were closely tied to the original PIM-style onboard applications, the File Logger to the telephony implementation (of which it was originally a component), the C Standard Library to Java (for which it was originally written to provide a minimal C wrapper for system calls), and so on. In all cases, these components have been part of the OS since its early releases.

The Cryptographic Token Framework and Certificate and Key Management components are relatively more recent, first appearing in Symbian OS v7. Their initial appearance in the platform represented the first steps toward providing complete and pervasive architectural support for secure network connections and secure browsing. The introduction of Platform Security, in Symbian OS v9, completes that process and takes it further,

providing a complete architectural solution to the problems of security, privacy and trust.

ANSI C and POSIX Support

The C Standard Library first appeared early in the evolution of the OS and has remained largely unchanged through subsequent releases, providing a basic subset of the standard ANSI C library functions and POSIX system calls. It is designed to make it easier to port programs written in C or mixed C and C++ from other platforms to Symbian OS, although it does not claim to create a complete POSIX-like environment on Symbian OS. Instead, it supports the essential library functions, for example `malloc()`, `free()`, `printf()`, and so on, that almost any C program needs in order to run. All of `stdio.h` and `math.h` are supported.

The goal is to solve the most basic problems of porting and to enable basic C programs to run, allowing developers to focus on porting specific program logic and mapping to Symbian OS native idioms. In many areas, the underlying operating system semantics (of POSIX and Symbian OS) are quite different. For example, native process and thread semantics, file semantics, console behavior, and error and signal handling are very different in the two systems.

The C Standard Library implementation was originally written to support the first Java port to Symbian OS and included the bare minimum of the library needed by Java. (The porting problem was exacerbated by the original licensing conditions for Java, which limited source code availability; providing a minimal POSIX support layer was the simplest solution.) Since then it has been used by other system components, as well as by third-party code, especially for porting programs originally written for Unix. For example, recent ports of Python rely heavily on it.[2]

While Symbian OS v9 improves the support for 'standard C', there is still no support for accessing native idioms such as active objects from the standard C library. In other words, the library does not attempt to provide a complete C language interface to Symbian OS. Thus while POSIX can be seen as a valuable migration tool, for complex system ports its omissions are significant. It is likely that future releases of Symbian OS will include increasingly complete POSIX support as part of the support for a more standard C and C++ application platform.

Secure Certificates, Keys and Tokens

The Certificate and Key Management framework provides a complete framework for managing and storing security certificates and keys, and supports certificate storage and retrieval, certificate-chain building and

[2] Nokia provides a Python implementation for S60 through ***http://forum.nokia.com***. Tim O'Cock's Python for Symbian OS can be found at ***www.monkeyhouse.eclipse.co.uk***.

validation, and key operations including importing and exporting RSA, DSA and DH key pairs. (There is no support for generating keys.) The framework is not generally available to third-party applications, but is used by system clients (Application Installer, for example) and licensee applications (browsers and VPN client applications).

The Cryptographic Token Framework provides the additional support needed to manage certificate or key-protected hardware tokens (media cards such as SD or Memory Sticks, for example), and again is available to applications as well as system clients. It also provides an API for displaying security-related dialogs to the user (the implementation is supplied by the user interface).

Tokens are used to store secure keys. The framework provides an abstraction based on stored keys or certificates or PIN-style key authentication, as well as finder support to identify and enumerate secure media. Typical uses of secure tokens include DRM-protected content on physical media or their equivalent software 'emulations' (for example, DRM-protected games or films), as well as downloads (for example, of music).

Both frameworks make use of the unified key store and unified certificate store, abstractions allowing devices to have multiple, coexisting, key and certificate store implementations and providing a single point of access for clients, regardless of where an actual certificate or key resides (e.g. it might be on external media).

The key store is a repository of private PKI keys and provides APIs for storing and retrieving keys and for managing the store itself. The certificate store is a repository of root and user certificates and it provides APIs for storing and retrieving certificates and for managing the store itself. Root certificates typically belong to a certificate authority. User certificates belong to and are authenticated by the phone owner, and are always associated with a private key which is stored in the key store.

The security-related components within the Generic Libraries collection sit between the application-level services (such as the secure installers and the Content Access Framework, which is a generic framework for controlling content access in a way which is transparent to applications) and the low-level implementation of the cryptographic libraries (see Chapter 10).

Other Generic Services

The Task Scheduler provides a mechanism for performing time-based or condition-based tasks by scheduling the launch of an appropriate application when the task trigger is met. (This is not a notification service therefore, it is an application launcher.) From Symbian OS v9.1, conditions may include Publish and Subscribe variables becoming true. Typical uses include scheduled connections (connecting to email or message services, for example) and scheduled backup or data synchronization. Note

that the Task Scheduler is a system server that always runs and which saves schedules to a permanent file store to ensure continuity across reboots.

Before Publish and Subscribe, the System Agent provided the means for storing and querying system state. From Symbian OS v9.1, most system-state values become Publish and Subscribe `RProperty` values to which clients can subscribe (given appropriate security-model capabilities). The System Agent retains only a few key services, for example, it defines and creates some default global system properties at startup and it maintains the Publish and Subscribe battery strength property.

The Event Logger provides an interface for logging and filtered querying of system events of interest to applications. Built-in and user-defined event types are supported. Typical uses are for creating call or message lists (a list of 'Recent Calls' in a phone application, for example). Events are expired when their lifetime is reached. However, the actual logging engine is optional and is supplied by the licensee (in the variant user interface on a particular device). If it is not present, calls to the logging APIs have no effect.

The File Logger, which provides a logging to file service, is deprecated in Symbian OS v9.1 and should be used only as a debugging tool. (It remains in the system for backwards compatibility purposes only.)

Component Collections

Components are organized into two small collections of servers, framework, and libraries. The common theme of the collections is general utility.

Generic Services Collection

This collection provides miscellaneous system services including some legacy components (retained for API compatibility).

- The Task Scheduler component is an application-launching server that supports creating, querying and editing of time- or condition-triggered tasks. From Symbian OS v9.1, clients should migrate to revised interfaces.
- The Event Logger component is only an interface (i.e. it is supplied only as a wrapper) supporting logging of events, for example, call

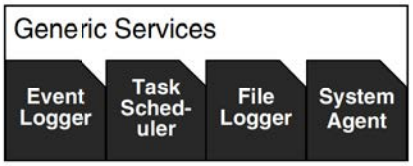


Figure 8.7 Generic Services components

Table 8.1 Generic Services Components

Component Name	Development Name
Event Logger	`LOGENGONGOING`
System Agent	`SYSAGENT2`
Task Scheduler	`SCHSVR_ONGOING`
File Logger	`FLOGGER, COMMSDEBUGUTILITY`

and message lists and retrieval, filtering and viewing by clients. The logging engine itself is assumed to be supplied by the variant user interface. If no engine is present, calls to the wrapper succeed but have no effect.

- The System Agent component is a legacy component that performed a number of useful functions for monitoring and reporting system state. From Symbian OS v9, the main System Agent functionality is taken over by the Publish and Subscribe service provided by the User Library (see the `RProperty` class). The System Agent retains a few key services only, for example, it defines and creates some default global system properties at start-up, and it maintains the Publish and Subscribe battery strength property.
- The File Logger component is a legacy utility for logging system or application messages to a log file. From Symbian OS v9 this is considered a debugging utility that is provided only for backwards compatibility.

Generic Libraries Collection

This collection provides system-level libraries for use by applications and system components.

- The Certificate and Key Management, Certificate Store and Key Store components provide a framework for certificate and key management that supports public key cryptography for RSA, DSA and DH key pairs (including storage and retrieval), assignment of trust status and certificate-chain construction, validation and revocation. Certificate

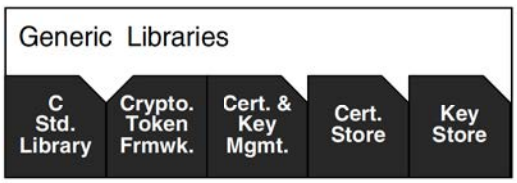


Figure 8.8 Generic Libraries components

Table 8.2 Generic Libraries Components

Component Name	Development Name
Certificate and Key Management	`CERTMAN`
Certificate Store	`CERTSTORE`
Key Store	`KEYSTORE`
Crypto Token Framework	`CRYPTOTOKENS, FILETOKENS`
C Standard Library	`STDLIB`

Store provides a single point of access for clients to certificates stored on the device. Key Store is a repository of private PKI keys that may be used to sign data, verify signatures, and so on, and provides APIs for storing and retrieving keys and for managing the store itself.

- The Cryptographic Token Framework supports the use of secure hardware tokens (i.e. encrypted media cards and file systems), for example DRM-protected games or films on SD cards or memory sticks, or their equivalent software emulations, for example, downloaded music tracks.
- The C Standard Library is a subset of the POSIX C library which maps C function calls in as simple a way as possible to native Symbian OS calls. It is a subset implementation and does not attempt to provide a complete POSIX environment on Symbian OS.

8.7 Multimedia and Graphics Services Block

Graphics has always been central to Symbian OS (see Figure 8.9), which was designed to support a sophisticated graphical user interface and sophisticated application graphics. In Symbian OS, there is no notion of character-based applications (except for test or development purposes); all applications are intrinsically graphical. Likewise, full-color support has always been an integral part of the OS design; even when running on 16-bit grayscale devices, 24-bit color modes were supported (which still remains beyond the capabilities of most phones).

Similarly, while the native font format is bitmapped (bitmap fonts are still preferred for small-screen devices, where pixel-perfect design is required to optimize for relatively small physical display size), support for FreeType vector fonts was introduced early on. Indeed sophisticated support for non-Roman fonts, including right-to-left and even bi-directional fonts, was always seen as central to the global aspirations of the OS.

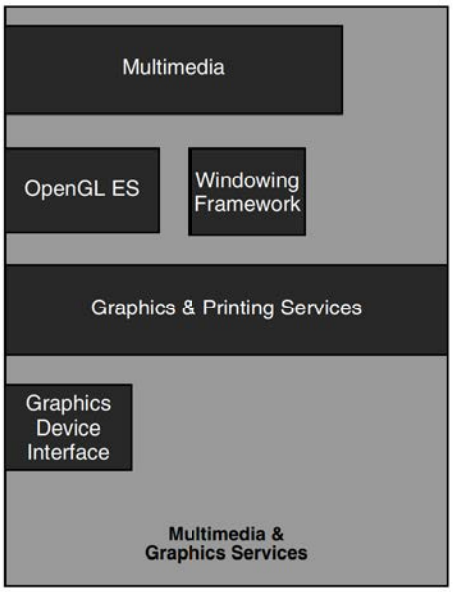


Figure 8.9 Multimedia and Graphics Services block

Audio data too was supported from the beginning, with a built-in recorder application forming part of the original application set for the system, something many phones still cannot match. (A Psion Series 5 running the early version of what became Symbian OS beat a cassette player hands-down for nailing that hard-to-master guitar lick.) Symbian OS moved onto phones when the state of the art was a polyphonic ring tone; in contrast, it allowed users to launch a complete sound clip as a phone ring tone (leading to offices full of baaing sheep and baby-gurgle effects).

Increasingly, as Symbian OS has driven into the phone market from its base as a more generic OS for the family of connected, PDA-like devices from Psion and as phones themselves have become more sophisticated, support for full-scale multimedia has become essential.

The Symbian-based Nokia 7650 was the first camera–phone outside Japan. The Symbian-based Sony Ericsson P800 played movie clips and the P900 shipped with a built-in MP3 player. The Symbian-based Nokia nGage integrated an FM radio along with game graphics and stereo sound.

The new device trends include full-stereo sound, multimegapixel cameras with true optics (true camera lenses and optical zooming) for both still and movie images, hardware-accelerated graphics, multiple high-pixel-density displays, and onboard high-definition broadcast TV.

Symbian OS v9 supports these hardware features with a new Multimedia Framework that is lightweight and flexible and aims to provide

consistent and hardware-independent interfaces at the application level, while providing flexible support to device makers wanting to integrate the available multimedia hardware and support new multimedia applications and services.

Graphics Design Goals

Graphics is central to the goals of the OS to provide an easy-to-use, consumer-oriented operating system capable of driving a wide range of devices but offering a sophisticated and, above all, open (to third-party application developers) platform for general application development.

From the beginning, the system has been optimized to produce fast graphics on low-power devices. The importance of rich font support was recognized early on, including support for exotic scripts. Increasingly, the focus for graphics has moved towards multimedia applications and games, making device graphics generally more critical, as well as specifically making it more important to support open graphics standards.

On the one hand, Symbian offers a much more integrated graphics architecture optimized for its device class than, for example, Linux, which requires licensing an application-level graphical toolkit, for example, Trolltech or GNU, on top of which to either license or implement a bespoke user interface. On the other hand, the Symbian graphics solution aims to be more carefully architected, more modular, and better-scaled than a system such as Windows Mobile, which has a monolithic user interface implementation (with its origins in the PC-centric, legacy design of Windows itself).

Multimedia Design Goals

The first implementation of a multimedia server was introduced in Symbian OS v7. It was enhanced and substantially re-architected in Symbian OS v8 and has evolved significantly in Symbian OS v9. Partly, its evolution is the result of the rapid pace at which multimedia hardware and services have migrated to mobile phones and the push from both licensees and operators to integrate sophisticated new hardware and support new media services, and partly it is a natural evolution enabled by other enhancements in Symbian OS (including the adoption of the real-time kernel, which opened the way for a significant change in phone hardware complexity, and platform security, which makes Symbian OS an ideal platform for a movie and music player, including DRM-protected media cards and downloads).

The Multimedia Framework therefore provides a single extensible framework for integrating support for audio, video, MIDI, automated speech recognition, cameras, and integrated broadcast tuners. Its purpose is to consolidate and standardize the multimedia APIs, so that they are

common across all devices based on Symbian OS, while also providing a flexible foundation for extension and customization.

The framework is designed around the concept of controllers that provide a full range of standard multimedia functions (such as audio and video recording and playback, as well as more advanced functions such as speech recognition) and define standard APIs, allowing uniform client access across all Symbian OS devices regardless of their different capabilities, and a standard plug-in interface.

The framework is implemented as a lightweight, multithreaded, ECOM-conforming plug-in framework, which may run as one or more threads in the client application process, and consists of a number of components that implement the application-level interfaces.

The actual implementation on any device is provided by controller plug-ins which are supplied by licensees and used by client applications to access multimedia functions. The media capabilities of a given device therefore depend on the available hardware and the supporting controller plug-ins, and are ultimately determined by the underlying device hardware. Multiple controllers may be available for any given format. Applications can choose whether they want to select a controller or to leave selection to the framework.

The Image Conversion Library provides an extensible plug-in framework, also conforming to ECOM, and a standard set of conversion plug-ins supporting conversions between standard image formats. The Camera component defines a standard API for onboard cameras and provides a reference implementation. The Broadcast Tuner component provides a standard API for onboard radio tuners.

The camera and tuner APIs are implemented as frameworks into which custom plug-ins are loaded to support specific hardware available on different Symbian OS devices but provide a standard API both for plug-in writers and for client applications, so that applications can work consistently across different phones including phones from different vendors.

The framework also defines the lower-level interface to the Media Device Framework (see Chapter 10), which defines device-level plug-in interfaces including support for hardware accelerators, and provides a standard set of device drivers.

The framework is heavily plug-in dependent. From a client application perspective, it provides a rich and consistent set of interfaces for all kinds of multimedia. From a system perspective, it provides a number of different plug-in interfaces to support writing of plug-ins that implement chosen levels of support (from logical to physical) for the onboard hardware.

By default, Symbian supplies only a simple audio plug-in, supporting WAV, AU and RAW formats. Codec implementations are provided for a number of encodings including various PCM encodings, A-Law and u-Law, and GSM6.10. Licensees are expected to supply the full set of

plug-ins required on a particular phone, providing custom controllers, codecs, and format support.

OpenGL ES

OpenGL ES is an open standard for 2D and 3D graphics, specifically targeted at embedded systems including consoles and phones. It defines application APIs for rendering, texture mapping, and other graphical effects, as well as a portable binding to native windowing systems, as a subset of the workstation- and desktop-oriented OpenGL standard.

OpenGL ES support in Symbian OS consists of a framework that implements the API binding and a standard client API definition but not a concrete implementation. A stub implementation is supplied by OpenGL ES Headers and the OpenGL ES component is a reference implementation of a third-party OpenGL ES renderer. The API includes Display Properties (optional in the OpenGL ES standard) that encapsulate drawing properties (e.g. displaying rectangles and clipping regions), enabling drawing to be delegated to threads that don't have access to an `RWindow` object.

The framework is provided to implement the OpenGL ES binding and ensure compatibility between different devices.

Windowing Model

The Window Server is at the heart of the graphics architecture of Symbian OS and it is central to the event-handling model that drives applications. Unlike many other operating systems, in Symbian OS there is no notion of character-based applications or devices (there are no teletypes or green-screen terminals in mobile phones). All applications in Symbian OS are intrinsically graphical and the screen is where application events are realized, as well as being an important source of application events. The Window Server is at the heart of screen control and is, therefore, central to applications.

The Window Server uses the concept of application-owned windows onto the display device to serialize access to the display by multiple concurrent applications. A window on a Symbian OS device is a rectangular screen region that can be drawn to on behalf of its owning application in response either to system or application events, and which receives focus events as well as keyboard and pointer (pen) events. The Window Server owns the screen as a resource and owns the single event queue through which all device events, whether system- or application-originated, are handled, managing kernel and application events as well as events generated by the Window Server itself and distributing them to applications or system user interface components (status bars and so on). The Window Server implements a classic Symbian OS pattern – serializing access to shared resources, which in this case include the physical display and

interaction and other events. (Note that devices may also have multiple physical displays.)

A window is an abstraction for making a screen region available to an application for interaction. A window abstracts a region of the physical screen. From the perspective of an application developer, a window is a screen region in which an application view can be constructed. To draw into the window area, and to receive user input, applications create controls inside windows and the controls become the units of interaction.

Applications are, by definition, window-owning processes. Applications may create and destroy windows, may have many windows, and may switch between them. Application windows form a window group. The first application window in a group is the top client window and an application must have at least one of these in order to display. Windows allow applications to display and have screen modes (e.g. color depth), a drawing area, and so on.

Logically, windows are maintained in a window stack, implying that they have a 'Z' order which is enforced by clipping; windows higher in the stack hide windows lower in the stack. Windows come in and out of focus (i.e. have the focus of the user) and, typically, the window at the top of the stack is the window which currently has focus. (A window group, however, may choose not to receive focus.)

By default, windows are the size of the full screen (less any area reserved for device status bars, control button arrays, or similar system-owned screen resources). Therefore, windows do not overlap but hide each other, a design optimized for small-screen devices (but a policy which is ultimately determined by the variant user interface running on a given device).

In implementation terms, a window is an 'R' class (`RWindow`) object returned by the Window Server to a client opening a window subsession in a Window Server session.

From an application perspective, most of the Window Server functionality is abstracted through the Control Environment and is made available to applications through the APIs provided to create and manipulate controls. Thus while applications need some knowledge of the windowing model, the window methods are provided through the Control Environment, not from the Window Server explicitly.

The Window Server is a system server that is started by the System Starter at boot time and runs until system shutdown. Only when the Window Server is running and providing access to the screen and events, can the Application and UI Frameworks be started, and only then can the phone application be run.

The Window Server is also responsible for starting some servers at startup and it provides the plug-in interface for animation (see Chapter 7). (It also provides other plug-in interfaces, for example the RSoundPlug-in interface. The Keyclickref plug-in is an example implementation of a

Window Server key-click plug-in library, which provides the audible clicks for keystroke events.)

The Window Server has a number of responsibilities:

- implementing client-side buffering of windowing commands to minimize calls across process boundaries between client and server, while enabling fine-grained control by the client (which can flush the buffer)
- managing bitmapped drawing via the Font and Bitmap Server
- managing the clipping and valid or invalid regions of screen, for example, when part of the screen becomes uncovered by some window
- managing system-initiated redraw events, window stacking (Z-order), etc.
- providing backed-up windows as a special case for applications that lack the ability to manage their own redrawing efficiently
- handling special effects including shadowing and animation.

Event handling is based around the `RequestEvent` active object, which the Window Server uses to get events from the kernel, once it has registered itself as the default event handler. Event types include digitizer and pointer events, keyboard events, and some other hardware events (including switch off, case open and case close, which are legacies of the early device architecture of the Series 5, a clamshell device in which closing the case caused the device to suspend and opening it caused the device to resume operation).

Window Server processes these events and passes processed events to clients. Thus, for example, pointer events may be translated into focus events or other logical events. Window Server may also perform rotation and other logical processing or scaling of screen coordinates for generated events (some devices support multiple screen orientations; others support full and 'flip' mode sizes). It performs key events and logical key events, including translations that are implemented by Front End Processor (FEP) plug-ins, for example to translate pointer events on an on-screen soft keyboard to logical keyboard events; to translate scan codes to character codes for physical key events; and to interpret hot-key events and combinations. Window Server also initiates events, for example redraw events, and manages the event queues for clients. Each client has its own queues.

The Window Server also supports a direct access (DSA) drawing mode, which bypasses the server itself but still enables an application to determine which screen region it owns (so that it does not overwrite other applications or system components which may have visible elements on the screen). The DSA framework notifies Window Server when it is invoked (but otherwise the Window Server is not directly involved).

The Window Server has been a central part of Symbian OS since the beginning but has seen many enhancements in subsequent releases, including semi-transparent windows, multiple screens, double buffering, and a configurable origin and scaling factor for windows (supporting rotated screens and flexible screen size).

Fonts and Bitmaps

From the perspective of the graphics system, all graphics devices are bitmap devices. All bitmapped graphics services and font services, including printing support, are managed by the Font and Bitmap Server. It owns the graphics devices and serializes client access to them (whether clients are applications or other system services). All access therefore to the screen or to printers and all bit-oriented screen operations, including font operations, are conducted through a client session with the Font and Bitmap Server, within a bitmapped device context. The Font and Bitmap Server also ensures that screen operations are efficient by sharing single instances of fonts and bitmaps between its multiple clients. It also provides the framework for loading bitmap and vector fonts.

While the Font and Bitmap Server owns the graphics devices, the Bit Graphics Device Interface (GDI) actually rasterizes drawing to bitmapped devices. Bit GDI implements the concrete instances of bitmapped graphics contexts, from the basic device abstractions providing hardware-independent access to display devices and screen attributes using a variety of graphics primitives. Bit GDI also provides transparency support (alpha blending).

Font management is delegated to Font Store, which manages all fonts in the system, both native Symbian OS format bitmapped fonts (glyph fonts) and open vector fonts, and performs closest-fit matching of font requests. Font Store provides APIs for storing, querying and retrieving bitmapped fonts and all properties of glyph fonts. Vector fonts are drawn by the FreeType Font Rasterizer and the available vector fonts are vendor-dependent. (Symbian OS includes a reference implementation of the FreeType rasterizer and reference bitmap fonts. Licensees may choose to replace the FreeType implementation with one of production quality or may omit it and replace the reference bitmap fonts with their own bespoke fonts.)

FreeType supports FreeType 2 TrueType fonts. On small-display devices, and on phones in particular, carefully optimized bitmap fonts offer a more optimal font solution than standard vector fonts.

WYSIWYG printing is provided by the Printer Driver Support printing framework, which stores and manages printer drivers and manages access to and mapping of printer ports, and for which reference implementations of concrete printer 'driver' plug-ins (type: PDR) are provided. Printer drivers are not device drivers in the standard sense of controlling physical

hardware; rather they are printer-driver information files that provide translations from device-independent bitmap-based graphics descriptions to printer page descriptions. More precisely, a printer driver implements the GDI-defined bitmapped device abstraction and is a DLL plug-in to the framework. Printer ports are virtualized over the available device hardware, typically serial or short link. As well as loading driver plug-ins, Printer Driver Support creates printer driver lists. WYSIWYG printing support is considered legacy functionality for a modern phone. (See the earlier discussion of application-level support for printing, for example, based on Bluetooth profiles.)

The close coupling of drawing and printing and the inclusion of support for line breaking and margin calculation alongside polygon and ellipse rasterization among the Bit GDI primitives shows the legacy of the early implementation of Symbian OS. For example, on a modern phone, fast rendering of games and smooth rendering of streamed video on a device where many other things may be going on (calls being received, music being played, and so on) is more relevant than margin calculation for office-style documents.

Graphics Contexts and Color Palettes

The lowest layer of the graphics system provides the abstract interface to the device hardware (the physical interface is managed by the logical and physical device drivers in the Kernel Services and Hardware Interface layer).

The GDI abstracts the physical graphics device (a bitmap display or raster device) as a Device Context containing settable drawing and font properties (pen and brush settings for line styles, character and font information and metrics), all drawing methods (for lines, polygons, circles, rectangles, as well as text and bitmaps), and the clipping region defining the drawable rectangle.

Since GDI pixels and font metrics are device dependent, methods are provided to map from twips values (Symbian OS device-independent units)[3] to pixels and to zoom fonts by a specified zoom factor.

Text rendering supports bi-directional text, that is, both right-to-left and left-to-right as well as mixed text, and line-breaking algorithms. GDI also manages color value, handling mapping RGB values into display-mode color spaces.

The Color Palette supports color-array handling and conversion between RGB values and palette indices, and supports dynamic palettes, that is, color palettes may be supplied by external classes, allowing clients to control the palette capabilities depending on the available device hardware.

[3] A twip is a decimal variant of a typographical point. A point is 1/72 of an inch; a twip is 1/20 of an inch.

Font metrics and selection (matching a device-specific font to the font request) were significantly improved in Symbian OS v9 to support higher-resolution screens and to better support screens with non-square pixels. Calculation algorithms for font metrics (ascender and descender sizes, capital heights, etc.) were added and there are methods that offer choices based on maximum height to guarantee that the supplied font fits the given screen space.

Graphics Architecture

At the heart of the graphics architecture are the Window Server, the Bit GDI and the GDI components. Together they provide the services required to write to bitmapped physical displays from within a system or application graphics context and support the windowing abstractions that allow multiple clients to independently manipulate the display.

The Window Server abstracts the key ideas of event-driven programming for graphical applications and applies object-oriented design principles (and native idioms of the operating system, for example, active objects) to provide a straightforward programming model for native applications.

From an application perspective, a window is a secondary object that is created from the application view (the top-level application control; every application needs at least one control that owns or controls a window, that is a view). Once associated with a view by a `Set()` operation, a window is abstracted to the top-level graphics context in which all subsequent drawing, clipping and similar operations are performed.

While the graphical architecture of Symbian OS is central to the user interaction and application model, so that, in effect, nothing can happen on a device (from a user perspective at least) without the involvement of the Window Server, from a system perspective graphics is well isolated from the kernel and the basic system services. Thus to implement a base port, a text-only version of the Window Server and a Text Shell replace the complete graphics infrastructure with a simple event handler and a console shell. The resulting bare-bones system has no application support, communications or other 'higher' services but, from a kernel perspective, it is fully functioning.

The graphics system therefore is (from the kernel perspective) just another user-side process; it runs user-side (i.e., in non-privileged mode) and uses the standard machinery of client–server inter-process communications to communicate both with the kernel (which is a server and to which it is a user-side client) and with its own clients.

The newer additions in the graphics area, for example, vector fonts and the OpenGL ES interface, as well as the Multimedia Framework itself, build on top of the basic window and graphics system.

Component Collections

Multimedia Collection

This framework defines application-level APIs for multimedia support of all kinds and provides a number of standard implementations as framework plug-ins.

Table 8.3 Multimedia Components

Component Name	Development Name
Multimedia Framework	`MMF, COMMON`
Multimedia Framework Plug-ins	`MMFAUDIOCONTROLLER, MMFRAWFORMAT, MMFAUFORMAT`
Image Conversion Library	`ICL, ICL_IMAGEDISPLAY, IMAGETRANSFORM`
Image Conversion Library Plug-ins	`ICL_GIFSCALER`
Camera	`ECAM`
Broadcast Tuner	`TUNER`

- The Multimedia Framework component provides a high-level extensible framework for multimedia support of all kinds, providing client utilities for common tasks, for example audio, tone, video, and MIDI playback and recording, as well as speech recognition. The framework is designed to accept controller plug-ins, which in turn provide the interface to lower level plug-ins (supplied by the Media Device Framework, see Chapter 10) that interface to hardware and provide acceleration APIs.
- The Multimedia Framework Plug-ins component provides controller plug-ins to the framework; reference controllers are supplied for standard audio formats.

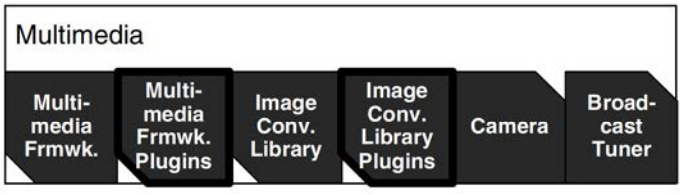


Figure 8.10 Multimedia components

- The Image Conversion Library component provides an extensible framework for integrating still-image conversion codecs into the Multimedia Framework. It recognizes picture file formats by providing a MIME-type recognizer plug-in to the MIME Recognizer Framework.
- The Image Conversion Library Plug-ins component provides default reference codecs for common still-image formats including GIF, JPEG, PNG, BMP and MBM.
- The Camera component provides an implementation for an onboard camera, allowing a camera object to be created and controlled and imagery data to be requested and received from it.
- The Broadcast Tuner component provides an implementation for an integrated broadcast tuner.

OpenGL ES Collection

These components comprise a framework supporting the OpenGL ES 2D- and 3D-graphics standard. OpenGL ES provides multi-client access to screen, keyboard, and pointer or digitizer for GUI applications and includes a keyclick reference plug-in that produces key or pointer clicks.

Table 8.4 OpenGL ES Components

Component Name	Development Name
OpenGL ES Headers	`OPENGLSHEADERS`
OpenGL ES Display Properties	`OPENGLESDISPLAYPROPERTY`
OpenGL ES	`OPENGLES9.X`

- The OpenGL ES Headers component provides standard OpenGL ES headers and binary definition files to encourage compatibility between OpenGL ES implementations for Symbian OS. The headers bind the OpenGL ES API to the underlying graphics model and support a plug-in renderer implementation.

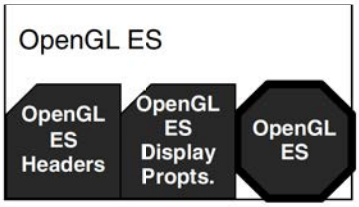


Figure 8.11 OpenGL ES components

- The OpenGL ES Display Properties component encapsulates display-drawing properties (e.g. display rectangles and clipping regions), enabling window surface access, that is, drawing, to clients from threads that do not own a window.
- The OpenGL ES component provides a reference implementation of an OpenGL ES renderer implemented as a plug-in, which is replaced by licensees.

Windowing Framework Collection

The Window Server owns and manages access to the screen as a drawable resource, which is made available to applications through the abstraction of windowed screen areas. It also provides access to the keyboard and pointer or digitizer for GUI applications, including the keyclick reference plug-in that produces key or pointer clicks.

Table 8.5 Windowing Framework Components

Component Name	Development Name
Window Server	`WSERV8.1`

Windows are at the top of the abstraction hierarchy for screen elements; all applications must own (or control) a window in order to display or to receive events. The Window Server receives and interprets events on behalf of applications, as well as generating events based on received application events (focus events, for example).

Graphics and Printing Services Collection

These components support all bitmapped graphics operations on display and printer devices, including all font and drawing operations. The principal components are the Font and Bitmap Server, through which all operations are made within a client-side server session to a bitmapped

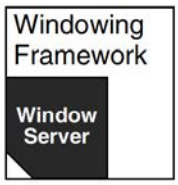


Figure 8.12 Windowing Framework components

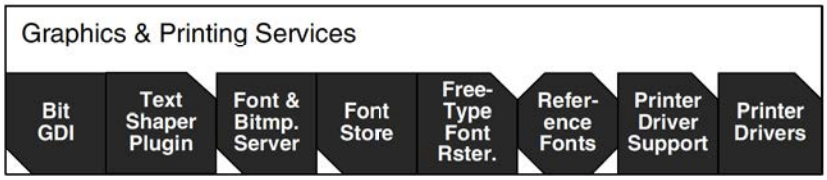


Figure 8.13 Graphics and Printing Services components

Table 8.6 Graphics and Printing Services Components

Component Name	Development Name
Font and Bitmap Server	FBSERV
Text Shaper Plug-in	ICULAYOUTENGINE
Bit GDI	BIT GDI
Font Store	FNTSTORE
FreeType Font Rasterizer	FREETYPE
Reference Fonts	FONTS
Printer Driver Support	PDRSTORE
Printer Drivers	PRINTDRV

graphics context, and the Bit GDI, which implements the bitmapped graphics context abstraction.

- The Font and Bitmap Server owns all bitmapped graphics devices and provides the framework for other graphics components. The server manages system-wide shared access to single-instance fonts and bitmaps, providing bitmap and font services for native bitmap fonts and vector fonts through its client-side APIs. It is responsible for loading the plug-in font rasterizer for vector fonts.
- The Text Shaper Plug-in component to the Font and Bitmap Server enables improved glyph placement for Hindi (i.e. Devanagari script).
- The Bit GDI component provides a polymorphic interface independent of device and display modes to bitmaps and the screen device via graphics primitives that implement the concrete device context for bitmaps.
- The Font Store component provides font storage and font file loading, using plug-in font rasterizer libraries if required. It also performs closest-fit matching of font requests.

- The FreeType Font Rasterizer component provides a reference implementation and library wrapper for the FreeType font rasterizer, supporting FreeType 2 TrueType font descriptions.
- The Printing Support component provides a framework that manages and loads printer drivers as bitmapped device context implementations and manages access to printer ports. It is considered a legacy component on most modern devices and is only relevant to PDAs.
- The Printer Drivers component provides reference implementations of concrete printer drivers that implement the polymorphic interface defined by GDI. It is considered a legacy component on most modern devices and is only relevant to PDAs.

Graphics Device Interface Collection

This is the lowest level of the graphics services, providing low-level graphics abstractions and color palette support.

Table 8.7 Graphics Device Interface Components

Component Name	Development Name
GDI	`GDI`
Color Palette	`PALETTE`

- The GDI component provides a device-independent graphics context abstraction, which supports drawing to various devices including screens and printers (which are treated as specialized graphics contexts). Normally all drawing, text display, and so on, is performed on a graphics context.
- The Color Palette component supports color-array handling, conversion between RGB values and palette indices, and dynamic palettes. Color palettes may be supplied by external classes, allowing clients to control the palette capabilities depending on the available device hardware.

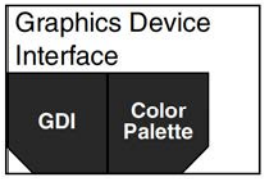


Figure 8.14 Graphics Device Interface components

8.8 Connectivity Services Block

Connectivity Services in Symbian OS (see Figure 8.15) consist of dedicated service and transport frameworks designed to support basic device or host connectivity functions, including backup and restore, remote file browsing, remote software installation, and so on.[4]

The first releases of Symbian OS based their connectivity on the proprietary PLP serial and infrared-based protocol. Symbian provided basic software for both PCs and devices, enabling backup and restore, synchronization of PIM-application engines, remote software install, and remote access to the file system. Licensees mostly provided basic customizations.

While Symbian OS v6.0 retained PLP, Symbian OS v6.1 moved to a TCP/IP-based framework (based on m-Router, licensed from Intuwave) and also introduced Bluetooth as a bearer, thus extending support to include cable, infrared and Bluetooth. m-Router also adds a service-loading framework and can load custom services.

From Symbian OS v8, there has been significant re-architecture of the Connectivity Services, principally on the host-side (in other words, on the host computer to which the device is connecting) but including the introduction of the Bearer Abstraction Layer to improve standardized access to connected phones.

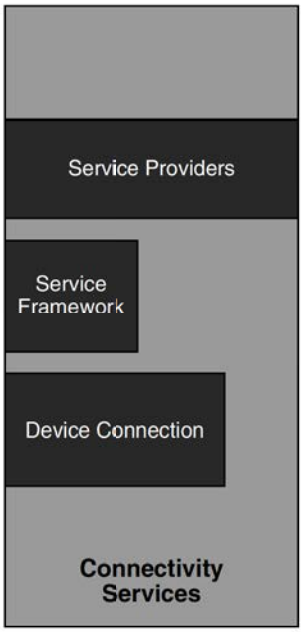


Figure 8.15 Connectivity Services block

[4] The best introduction is [MacDowell 2005].

Design Goals

Good connectivity is a vital feature for any mobile device and especially for consumer-oriented devices. Symbian OS provides good device-side support for generic connectivity services based on configurable, standards-based technologies (such as SyncML), and the drive towards more consumer-oriented devices will hopefully see licensees (or opportunistic third-parties) providing good solutions for connecting to all host platforms, including Macintosh and Linux, for example. Easy connectivity based on standard technologies and compatible between devices from different licensees across multiple host operating systems is vital to support migration of data between devices (from an old device to a new device, for example).

Interestingly, while Symbian OS makes TCP/IP the standard protocol for its connectivity services, OBEX is more common on phones not based on Symbian OS. OBEX is optimized for simple transfer of small objects, for example, contact records and SMS messages. While OBEX is supported by Symbian OS and while some licensees may provide their own support for OBEX-based connectivity, it is not part of the standard connectivity solution.

Overview

The connectivity architecture provides a framework within which the device-side of TCP/IP-based device-to-host services can be created. Since the actual bearer is abstracted, such a service runs on any bearer. Implementations are provided for the basic device–host connectivity services of device backup, remote software installation and remote file browsing.

Windows PC desktop-side implementations are supplied as part of the Connectivity Services implementation but, in principle, the services on the Symbian OS device are agnostic about the host operating system. Since the services are based on TCP/IP, host-side implementations can be written for any operating system. Typically, all licensees provide a host connectivity suite of some kind; most support only Windows, some support Mac OS/OS X. Third-party freeware packages provide varying degrees of support for Linux or Unix connectivity for Symbian OS devices.[5]

The device-side framework is extensible, so that new (device- and host-side) services can be written, and open, so that host-side services

[5] ***http://symbianos.org/~malm/SymbianLinuxHowTo.html*** documents connectivity solutions for legacy releases up to Symbian OS v7.0; for current Symbian OS phones, data synchronization with other SyncML supporting systems should be possible but may require configuration. Alternatively, ***www.scheduleworld.com*** provides a web-based SyncML service, which should enable synchronization between Symbian OS and other SyncML-supporting systems.

can be written for platforms (e.g. Linux/Unix) that device vendors do not support out of the box. The framework is intended for use by developers of host-side software to access the device and its applications and is customizable by extension.

As supplied, the PC-side connectivity application uses Windows Winsock over RS232 serial, USB, Bluetooth, and infrared connections. On the device-side (i.e. Symbian OS), the chosen bearer propagates (through the Sockets Server) to a Connectivity Services Server Socket. Bearer-level components interoperate with the Sockets Server (see Chapter 9) to provide services to the framework.

Services

Connectivity Service Providers are device-side services that support basic interactions with a desktop host to perform device backup to the host, file browsing and transfer (in both directions; typically, browsing the device file system from the desktop and copying files between the device and desktop host) and software installation (from desktop to device).

The basic supported services are:

- backup and restore of a drive on the device to a desktop host
- file management (e.g. copying files to and from the device, renaming and deleting files and directories on the device, and formatting device drives)
- installation of software from the desktop host.

Additionally, the infrastructure supports starting named services on the device from the desktop host and managing the connection between the device and the host.

Data synchronization functions are not supported by the Connectivity Services but are provided elsewhere (for the device side, see Chapter 7; on the host side, there are various third-party offerings as well as licensee-provided software packages).

All the supplied services use the Service Broker framework. The Remote File Server provides an interface, via the Service Broker, to the device file system for a host-side client. Similarly, the Software Install Server enables a host-side client to interact with the device Application Installer to install SIS, JAR and JAD files over TCP/IP or OBEX. Similarly also, the Secure Backup Server enables a host-side client to interact with the Secure Backup Engine, which performs the interaction with the device-side file system and other processes to back up data from the device to the host.

Framework and Transport Abstractions

The Service Broker framework is the core of the Connectivity Services implementation, allowing device-side services to register a port number

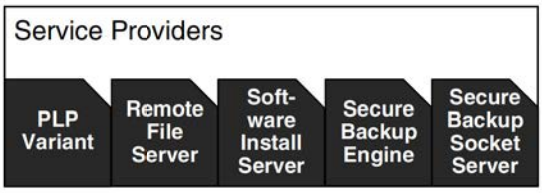


Figure 8.16 Service Providers components

for use by host-side clients, allowing host-side services to be started. The Service Broker protocol requires a TCP/IP connection to the host, for which it relies on the Bearer Abstraction Layer (BAL). Named services (supplied by the connectivity component) use the Service Broker. Port-number registration is based on XML-defined configuration files.

The Bearer Abstraction Layer (introduced in Symbian OS v9) provides a bearer-abstraction framework and a connection-management API to PC-link-type applications, allowing selection and configuration of connection bearers. Typically, the link application is provided by a licensee as part of a connectivity suite for a particular product. The framework supports plug-ins that encapsulate actual bearers (for example, m-Router).

Server Socket is a helper library that allows TCP/IP services based on port numbers to be created for use by the Service Broker, which is simpler and more ROM-efficient than creating bespoke named services from scratch.

Architecture

The Connectivity Services block provides device-side support for connectivity services. Services are organized around a central Service Framework component, the Service Broker, with named services, which are clients of the framework and use it to propagate service port numbers to remote clients, and bearer services, which are used by the framework to provide TCP/IP-based services over a variety of available bearer technologies.

The Bearer Abstraction Layer provides a common platform on top of the m-Router TCP/IP-based transport, independently of the actual bearer. Bearer support is provided to the Bearer Abstraction Layer as a plug-in, interfacing to a networking Sockets Server socket connection.

Component Collections

Service Providers Collection

These components provide named services which run on the device side to provide service interfaces to remote (host-side) clients. All use the Service Broker as an intermediary to propagate their port numbers to the remote client.

Table 8.8 Service Providers Components

Component Name	Development Name
Remote File Server	`REMOTEFILESERVER`
Software Install Server	`SWINSTALLSERVER`
Secure Backup Socket Server	`SBSERVER`
Secure Backup Engine	`SECUREBACKUPENGINE`
PLP Variant	`PLPVARIANT, PLP, BRDCST`

- The Remote File Server component provides on-device file-management functions to a remote client over TCP/IP, including access to backup and restore functions provided by other system components.
- The Software Install Server component interacts with the software installation components on the device to enable remote installation of SIS, JAR and JAD files over TCP/IP or OBEX. Installation events can propagate to a connected host, passing progress information and errors and allowing user interaction.
- The Secure Backup Socket Server component provides backup/restore functions to a remote client over TCP/IP.
- The Secure Backup Engine component manages backup and restore of device-side data, including private data and installed software, as controlled by the Secure Backup Socket Server. This component exposes an API and can be used by other components to carry out a remote backup and restore (for example, to a connected PC) or a local backup and restore (for example, to a removable memory card).
- The PLP Variant is a deprecated legacy component that returns fixed-device information, for example the device ID and required free memory, to applications running on other devices or connected hosts. It is retained only for compatibility with third-party components that use some of its APIs. It is implemented as a DLL to which applications link, not as a plug-in.

Service Framework Collection

This service based on configuration files and port registration enables device-side services to register a port number for use by PC-side clients, which can query for and start device-side services. The configuration files have an XML-based format.

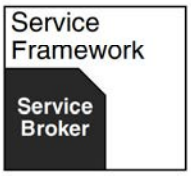


Figure 8.17 Service Framework components

Table 8.9 Service Framework Components

Component Name	Development Name
Service Broker	SERVICEBROKER

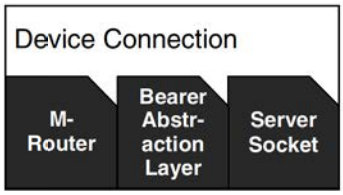


Figure 8.18 Device Connection components

Table 8.10 Device Connection Components

Component Name	Development Name
Bearer Abstraction Layer	MROUTER-PLUG-IN
Server Socket	SERVERSOCKET
m-Router	MROUTERSECURE

Device Connection Collection

This is the lowest (bearer-level) layer of the phone's Connectivity Services.

- The Bearer Abstraction Layer component is a framework for plug-ins, which encapsulates actual bearers (for example m-Router), providing a connection-management API to PC link-type applications.
- The Server Socket component is a helper library that supports creating (new, unnamed) port-number-based TCP/IP services for use by the Service Broker for device–host communications, for example with a PC. It communicates service port numbers and manages messages and commands.
- m-Router is a licensed, PPP-like data-communications protocol and framework, which provides a TCP/IP-based connection between two

devices (typically, a Symbian OS device and a desktop computer; it runs on both sides of the connection). The connection may run over Bluetooth, infrared, USB, or serial cable connections. m-Router provides a proprietary framework for loading custom services.

9

The Comms Services Block

9.1 Introduction

The system model represents Comms Services as a major, self-contained block within the OS Services layer of Symbian OS.

'Comms' (or communications), in this context, really means 'data communications' – the art, science and technology of moving data between different devices over direct connections or networks. See Figure 9.1.

What connections are available depends both on the hardware architecture of a given device and on what services happen to be accessible through the hardware at any particular time. A typical modern mobile phone includes a data cable connector of some kind for connecting to a desktop computer (typically for data synchronization and backup), infrared or Bluetooth radio or both for more transient connections (to other phones or devices such as printers) and, of course, the telephone radio hardware itself. Typically the data connector is proprietary but, increasingly, mini-USB ports have begun to appear on phones. On devices such as PDAs they are standard, as they are on other digital devices such as cameras and music players. Most recently, Wi-Fi has begun to appear on high-end phones.



Figure 9.1 The Comms Services block within the OS Services layer

Whatever the physical connections and whatever their purpose, the issues from a communications perspective are essentially the same.

- Two-way communications requires *protocols*; to successfully exchange data requires a surprisingly complex set of shared assumptions between two parties: getting the other party's attention, agreeing who speaks when, agreeing what counts as 'data', keeping up with each other, and so on.
- As well as protocols to manage the conversation between the end parties, protocols are required to relay or *transport* data between them, if the two parties are not directly connected to each other.
- Finding a *route* that connects the parties can also be complex; even where the parties are directly physically connected, an appropriate interface to the connection must be selected and configured (there may be multiple interfaces available, even for the same physical connection) and where there is no direct connection, a network route must be found.
- Specialized hardware requires appropriate *drivers*, to push data through and manage the hardware state (powering hardware down when not in use is especially important in a low-powered or battery-powered device, for example).
- In a multitasking system, contention for hardware between multiple clients is likely so that hardware needs to be *shared* (for example, if two applications are trying to use a serial port at the same time).
- Finally, at the application level, *settings* may need to be saved, shared, updated and managed.

Communications is complex because the task is complex. Communications also continues to evolve at an explosive rate, not least because it is also where computing and telephony converge, where wired and radio technologies converge and where personal and enterprise usages converge. For all these reasons it is usually considered to be at the technological leading edge.

Symbian OS supports a wide range of communications technologies including conventional serial communications, short link technologies such as USB, Bluetooth and infrared, as well as networking technologies, from standard Internet protocols to newer protocols such as SIP (which are designed to support services from VoIP to the latest packet-based data services) and, of course, telephony voice, data and messaging services for 2G, 2.5G and 3G networks, whether GSM/UMTS or CDMA/CDMA2000.

Symbian's communications support has evolved not just to track new technologies but also in response to their rapid convergence and, in particular, in response to the increasing importance of packet-based technologies for 2.5G and 3G telephony services.

9.2 Purpose

Comms Services in Symbian OS provides the support for a wide variety of communications protocols and services:

- Serial protocols including RS232, IrDA and USB
- Bluetooth radio
- Networking protocols including TCP/IP (both IPv4 and IPv6), network security (TLS and IPSec) and dial-up protocols (PPP and SLIP)
- Wi-Fi
- 2G, 2.5G and 3G mobile telephony voice, data (including fax) and messaging services for GSM/UMTS and CDMA/CDMA2000 networks.

These protocols in turn enable the infrastructure for higher-level services including:

- networking including browsing and VPN support
- SIP session support
- email, SMS, MMS, WAP and OBEX messaging
- SyncML data synchronization
- WAP browsing
- Fax.

These services are supported over physical hardware including cable serial ports, infrared, USB connectors, Bluetooth radio and GSM/UMTS or CDMA/CDMA2000 phone–air interface.

The system model divides the Comms Services block into four distinct sub-blocks: the Comms Framework, which provides the overall supporting infrastructure for data communications, and Telephony, Short Link and Networking sub-blocks, each of which defines the dedicated services required for its respective technology.

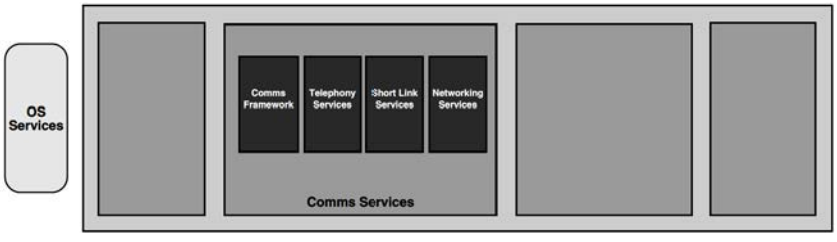


Figure 9.2 The Comms Services sub-blocks

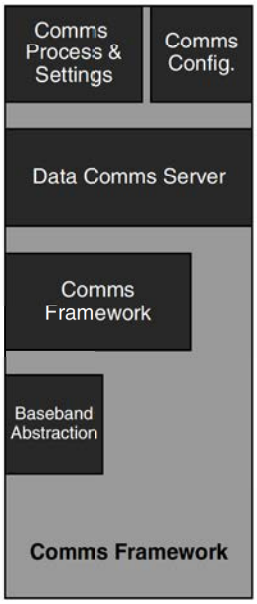


Figure 9.3 Comms Framework sub-block

Comms Framework

The Comms Framework provides the generic infrastructure that supports all communications services.

Most importantly, it includes the Comms Root Server, which is the 'meta' process server for all communications services and the ESock Socket Server which provides the generic, sockets-style interface used to access all communications services. See Figures 9.2 and 9.3.

Telephony Services

The Telephony Services are based on the ETel Telephony Server (and its extensions) that provides support for 2G, 2.5G and 3G mobile phone networks, including GSM/GPRS/EDGE/UTMS (2G/2.5G/3G) and CDMA/CDMA2000 (2G/2.5G/3G North America).

GPRS and EDGE are the incremental packet data and 'go faster' increments to GSM; UMTS and CDMA2000 are the respective GSM and CDMA evolutions to 3G. See Figure 9.4.

Networking Services

Networking Services provides packet-based network services with Ethernet emulation and includes the TCP/IP stack implementation, secure

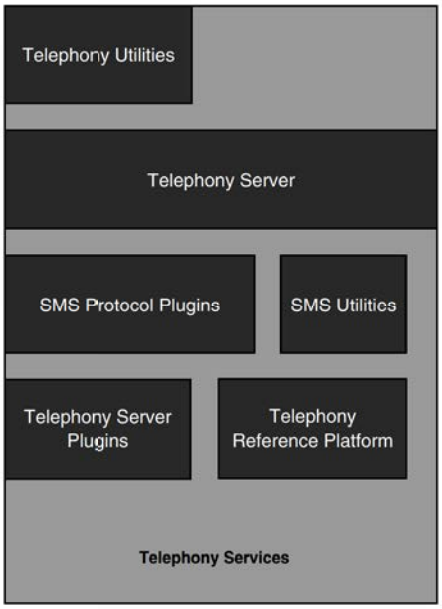


Figure 9.4 Telephony Services sub-block

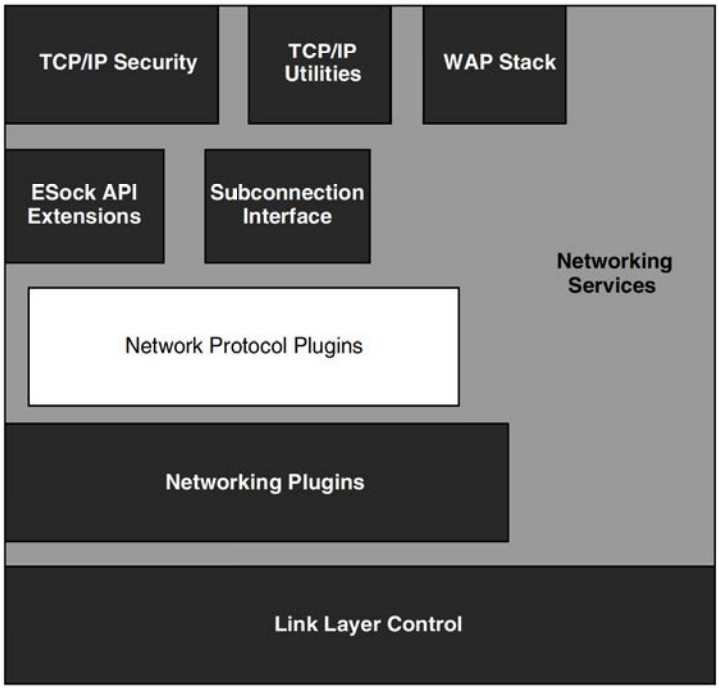


Figure 9.5 Networking Services sub-block

networking extensions including TLS/SSL and IPSec, which support secure browsing and VPN gateways, together with a variety of application-level Internet services including FTP and HTTP. (FTP does not expose public APIs.) All networking services are designed to be virtualized over telephony, serial or short-link bearers.

Support for Wi-Fi appears for the first time in Symbian OS v9 (although licensees have introduced Wi-Fi-enabled phones based on earlier releases). See Figure 9.5.

Short-link Services

Short-link services provides USB, Bluetooth and infrared services including support for the OBEX binary object protocol, USB class support that enables a Symbian OS phone both to use and serve as a USB host, and full implementations of the IrDA and Bluetooth protocol stacks. See Figure 9.6.

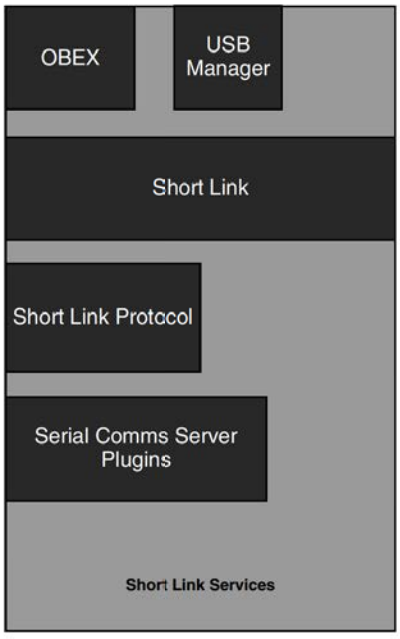


Figure 9.6 Short-link services sub-block

9.3 Design Goals

A phone is an extreme case of a mobile, connected device, which was the original design point for Symbian OS. While the ER5 release was explicitly targeted at PDA-style devices, even as the first Symbian OS devices reached market, convergence with mobile telephony was beginning to

drive the company strategy. Symbian OS has been a leader in the trend which has seen PDA functions largely absorbed into mobile phones.

Compared with the original Symbian OS devices, current mobile phones (even low-end ones) make vastly greater demands on communications support. On the Psion Series 5, for example, the communications hardware consisted of a single UART, which could be switched between the serial port and the infrared port but could not be used by both simultaneously. Despite the simple hardware, a full set of integrated communications applications was envisaged, from email and web clients to network news readers and multiplayer games (network Doom, for example) and, of course, including infrared printing and beaming.

By the time the Series 5 came to market, communications support in the operating system had already been extended to include basic telephony. However, following the logic of the simple communications hardware design of the Series 5, the early networking and telephony use cases envisaged a Symbian OS device as one half of a two-box solution, using a conventional serial modem or a GSM mobile phone as a dial-up modem to connect to an ISP for network access (including Internet) or driving a GSM mobile phone (sending AT commands over a serial link), for example to dial directly from a phonebook on the Symbian OS device or writing and sending SMS messages from the Symbian OS device via a GSM phone. In each case the physical link was serial (either cable or infrared).

Even when Symbian OS migrated onto devices with onboard phone hardware (even before the release of the Series 5 in July 1997, phone projects were underway with licensees), the connection between the phone-side hardware (a dedicated second processor running a GSM stack) and Symbian OS was serial. Thus even true telephony functions, such as setting up lines and answering and making calls, ultimately went through the serial server and a serial port.

Each subsequent release of Symbian OS has taken it a further step away from this early legacy to support the evolving reality of data-enabled phones capable not just of full network access (browsing or email, for example, over a VPN tunnel into a company network) but also of running real-time communications applications, for example, video conferencing which requires two-way, real-time video streaming.

As Symbian OS has evolved, it has become capable of real-time processing and thus capable of directly hosting the telephony baseband stack, making single-core phone designs possible. (The real-time kernel first appears as an option in Symbian OS v8 and is standard in Symbian OS v9.)

In Symbian OS v8 and Symbian OS v9, Comms Services has evolved significantly. In particular, the Root Server was introduced as the primary communications server, responsible for starting and stopping the dedicated communications servers on demand and providing the common context within which all communications servers run. (In earlier releases, the C32 serial server provided this service.) The goal is to support more

seamless interoperability between services and the faster data throughput required by new high-data-rate services.

In another significant change, from Symbian OS v9 the Comms Database has been integrated into the Central Repository, which provides a single point of storage for all system settings and a single common interface to all settings and service configuration. (The legacy CommDB interface is retained as a 'shim' layer providing backwards compatibility for existing applications.)

9.4 Overview

Symbian OS is designed for devices that typically do not have permanent or predictable connections (unlike a networked desktop computer, for example) and which have also typically not had even transient Ethernet connections (although this is beginning to change as Wi-Fi starts to appear on phones). The key requirement for communications services is therefore the ability to virtualize almost any service required at the application level over whatever transient connections are available at the time.

The hallmark of Symbian's communications implementation is the high degree of integration between the services at application level and the high degree of interoperability of technologies at a system level.

Logically, Comms Services is divided into sub-blocks, based on technologies. Each sub-block is organized around one or more primary servers and frameworks. Each server exposes client interfaces through its client-side APIs; implements system-level services by providing appropriate protocol implementations as Socket Server plug-ins; and defines a hardware adaptation interface through a framework for which it provides implementation plug-ins, while also enabling extension by licensees and partners (who can write their own plug-ins to support bespoke hardware).

9.5 Architecture

Servers and frameworks, characteristically for Symbian OS, provide the unifying architectural patterns for Comms Services.

The servers collaborate to provide the necessary level of interoperability essential for mobile devices which, by definition, rely on transient connections of varying kinds, depending on availability, rather than being a permanent part of a known, fixed infrastructure.

Each server exposes a client-side API. Each server is implemented as a framework. The frameworks supply the mechanisms for extensibility, which is designed in at a number of levels (see Figure 9.7):

- new protocols can be added to the system by server extensions
- new hardware types at the lower level can be supported by adding

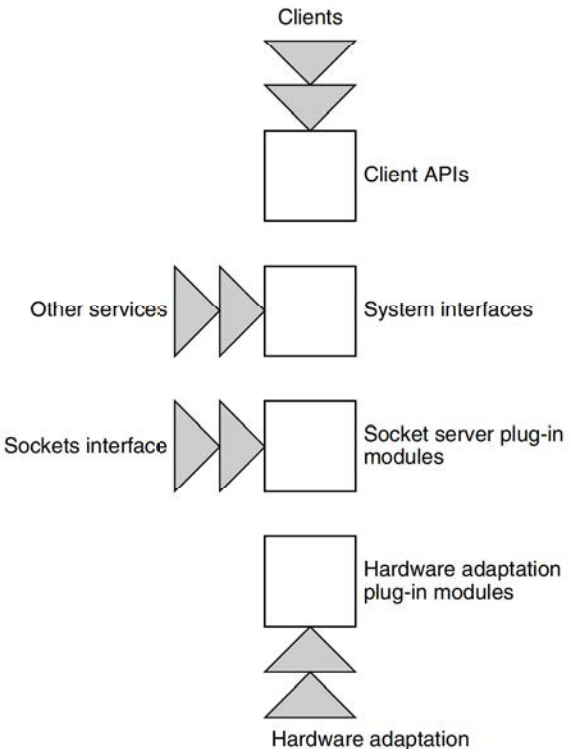


Figure 9.7 Logical layering of Comms Services

supplier module extensions (plug-ins which provide the hardware abstraction).

The architecture has proved its flexibility and adaptability over time, as it has evolved to support technologies such as Bluetooth and USB, as well as almost continuous evolution in telephony.

The Comms Server Model

In Symbian OS, each dedicated communications service is organized around a principal server and a protocol implementation. The servers include the ETel Telephony Server that provides telephony services, the C32 Serial Server that provides data communications services (typically virtualized over short-link connections), Internet extensions to the ESock Socket Server that provides networking services, and Bluetooth and USB managers that provide short-link services.

In the original communications architecture, all communications services were virtualized over a simple serial connection, supported by a hardware architecture which provided a single UART switchable between

a cable serial port and an infrared port. The C32 Serial Server was therefore the primary service provider, accessed directly by clients using its client-side APIs. The Serial Server also provided the framework which defined the low-level abstract API for communications modules (CSY files) which were implemented as plug-ins supporting the available serial hardware (the serial port and IrDA).

Networking services were designed around a server that provided a Berkeley-style Sockets API and a TCP/IP stack implementation, which was loaded as a server plug-in. At a lower level, however, all networking services were virtualized over serial connections (for example, an IrDA link to a network-connected computer).

When telephony services were introduced to the operating system, the design was quite closely modeled on the serial services architecture, with a primary server, the ETel telephony server, providing the client-side APIs and the abstract framework for hardware-facing telephony modules (TSY files), which were analogous to C32 Serial Server CSY communications modules. Interestingly, the addition of telephony services did not substantially change the earlier assumption of the primacy of serial communications, since the initial expected use for telephony was a two-box solution, using a serial connection (either cable or infrared) to connect a Symbian OS device to a modem or a mobile phone.

This was more or less the communications architecture of the first releases of the operating system and largely survived through the Symbian OS v6 and Symbian OS v7 releases. Over those releases, there were significant extensions, most obviously to telephony and networking to add the required packet capabilities for 2.5G and 3G data services, as well the addition of new short-link technologies such as Bluetooth and USB. However, the general principle of the serial server as the primary communications server (the 'first among equals') remained even though, as each release increasingly specialized the operating system for mobile phones, the primary communications use case was not serial communications but on-board telephony, with or without networking.

This architecture was unsatisfactory for a number of reasons, not least of which was the resource cost of having the serial server running all the time to support non-serial communications services.

Beginning with Symbian OS v8, therefore, some significant changes to the communications architecture were introduced as the foundation for further evolution to support the increased demands for high data rates. The key change was to introduce a primary communications server, the Comms Root Server, which is designed to provide a purpose-built, lightweight server for which the dedicated communications servers (C32 serial, ETel telephony and ESock sockets servers) act as service providers. The Serial Server is relieved of its privileged role and becomes just another dedicated service provider.