The BeOS BIBLE

SCOT HACKER

WITH HENRY BORTMAN AND CHRIS HERBORTH

Peachpit Press

The BeOS Bible

by Scot Hacker
with Henry Bortman and Chris Herborth

Peachpit Press
1249 Eighth Street
Berkeley, CA 94710
(800) 283-9444
(510) 524-2178
(510) 524-2221 (fax)

Find us on the World Wide Web at: **www.peachpit.com**

Peachpit Press is a division of Addison Wesley Longman

Copyright © 1999 by Scot Hacker

Notice of rights
All rights reserved. No part of this chapter may be reproduced or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher. For more information on getting permission for reprints and excerpts, contact Gary-Paul Prince at Peachpit Press.

Notice of liability
The information in this chapter is distributed on an “As is” basis, without warranty. While every precaution has been taken in the preparation of this chapter, neither the author nor Peachpit Press shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this chapter or by the computer software and hardware products described herein.

Trademarks
Be, BeOS, and the Be and BeOS logos are registered trademarks of Be Incorporated in the United States of America and other countries. All other products and company names mentioned in this book may be trademarks of their respective owners.

ISBN: 0-201-35377-6

The Kits

Behind the Scenes • 3
- *The Application Kit* • 6
- *The Storage Kit* • 8
- *The Interface Kit* • 10
- *Kernel Kit* • 13
- *Media Kit* • 15
- *Network Kit* • 17
- *Game Kit* • 17
- *The Support Kit* • 18
- *The Translation Kit* • 18
- *The Mail Kit* • 19

Chapter Summary • 20

As described elsewhere in this book, BeOS is an object-oriented operating system, from its lowest levels to its highest. Wherever possible, code is reused and recycled, rather than re-created from scratch. Each of the system servers described in Chapter 5 dips into its own pool of shared code and doles it out to applications as needed. These pools are divided into broad categories of functions called "Kits," and provide fundamental services (like networking, audio, data translation, and file management) to the system in the cleanest, most efficient way possible. BeOS includes a total of ten Kits; their functions are explored in this chapter.

Behind the Scenes

While the collection of BeOS "Kits" is of interest mainly to programmers, many technically oriented end-users may enjoy and benefit from this "glimpse behind the scenes." This chapter is optional reading for users interested in reaching a deeper understanding of the way BeOS operates, and as a primer for users interested in learning to program BeOS applications. It summarizes information available in The Be Book (on your system in HTML format at /boot/beos/documentation/BeBook/) and in these more extensive reference manuals:

The Be Developer's Guide. The Be Development Team
1997, O'Reilly and Associates. ISBN 1-56592-287-5.

Be Advanced Topics. The Be Development Team
1998, O'Reilly and Associates. ISBN 1-56592-396-0.

If you've got two applications open at once and both of them include scrollbars, pulldown menus, resizable borders, and title bars, there's no need for both of them to include all of the code necessary to construct those objects, is there? Of course not. The codebase that describes the rudiments of a basic application is stored at a central location in the operating system. In this way, BeOS achieves maximum efficiency and minimum redundancy, while application developers enjoy the luxury of not having to worry about writing all that stuff themselves. This object-oriented arrangement also ensures a consistency of behavior and appearance throughout the system so that all of your applications look, feel, and behave basically the same way.

Managing this interplay of services and the code that enables the services are two interrelated constructs: the system servers and the system Kits. You never see the servers or Kits, as they don't have an interface and there's nothing user-configurable about them. The servers (introduced in Chapter 5, *Files and the Tracker*) are active programs (or "daemons") that lurk in the background,

Figure 1

The BeOS system servers have no user interface and are not user-configurable—they lurk in the background awaiting requests from the rest of the system. When a request is received, they spring into action. Some of the servers pictured here, such as the app_server *and the* registrar*, are very active and are involved in almost everything you do. Others, such as the* syslog_daemon *and* print_server*, are seldom called upon. The system servers function sort of like liaisons, intermediating between the system Kits and your running programs.*

always running, waiting for requests from other programs before springing into action. The servers are compiled, running programs—you can see their icons by opening up your /boot/beos/system/servers folder in the Tracker.

The Kits, on the other hand, are not programs at all, and they don't "run" in the background. Rather, they're collections of objects classified by the jobs they do. The concept of "objects" here is a little abstract. In reality, the Kits are defined in shared code libraries that are called upon by all programs, including the servers. There are ten Kits in all, and they collectively define the application programming interface (API) that programmers use to make their generic C++ code talk to and work with the specific aspects of BeOS. They are:

- The Application Kit
- The Storage Kit
- The Interface Kit
- The Kernel Kit
- The Media Kit
- The Network Kit
- The Game Kit
- The Support Kit
- The Translation Kit
- The Mail Kit

You'll notice a certain amount of overlap between the list of Kits and the list of servers (for example, there's an Application Kit and an app_server, a Mail Kit and a mail_daemon). But there are also a number of Kits not associated with a server, and vice versa. The relationship is not one-to-one because some of the Kits serve umbrella functions for all of the servers. For instance, the Storage

Kit handles the nitty-gritty of reading from and writing to your disks—a job that most of the servers need done in one way or another. Likewise, a Game Kit is necessary for reasons we'll get into later, but games don't require their own server to function. In other words, some Kits are important to all applications, while others come into play only in certain instances. All GUI applications need windows, borders, scrollbars, and menus, so all GUI apps draw on the Application and Interface Kits; but only applications that deal with audio/visual data need to draw on the Media Kit. Finally, all of the servers "sit on top of" the system kernel, interfacing with it directly.

So how are all of these things related? It sort of depends on your point of view. As an end-user, it may make the most sense to think of the servers as sitting between the kernel and your applications. Servers don't serve you—they serve your applications and your applications serve you.

From a programmer's point of view, however, it's a little different, as developers don't access the servers directly. They access the Kits from within their code; the Kits in turn know how to talk to the servers. A programmer doesn't say, "Hey network—wake up!", but something more like, "I need a function that can establish a connection to another machine on the network." Since the function called is part of the Network Kit, it automatically establishes communication with the Network Server.

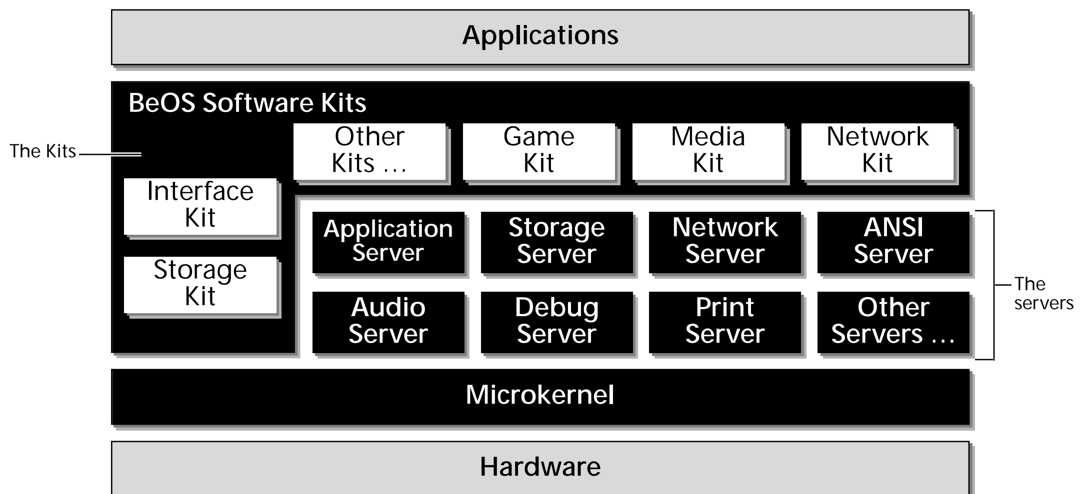


Figure 2

The servers sit between the kernel and your applications, functioning as an adjunct to or extension of both. Meanwhile, the servers look to the Kits as a codebase—a set of rules to follow in any given situation. For example, when you click on a link in NetPositive, the browser sends a message to the net_server *requesting network services so it can do its job and bring back the requested page. The* net_server *operates according to rules set forth in the Network Kit, which defines the structure and programming interface of all network operations. The heavily threaded servers are capable of handling any number of applications simultaneously, which is the essence of the client-server model introduced in Chapter 1.*

Each Kit consists of a collection of "header" files—raw C or C++ code written by Be engineers. When a programmer needs a system service, she invokes a Kit by creating an appropriate object or "including" the appropriate header file and invoking the appropriate function.

If you'd like to study more technical documentation on BeOS's system Kits, it's right there on your hard drive, at /boot/beos/documentation/BeBook/. There you'll find a collection of folders, each named for a separate Kit. Each folder contains a series of HTML documents describing that Kit in detail (though not always in plain English).

Digging through the Headers

Unlike the Windows or MacOS worlds, where end-users may work for years without ever seeing a scrap of actual code, many BeOS programs are distributed with full source code, and the operating system installs with a full collection of headers and libraries, in addition to a free and unlimited application development environment. As in the Unix world, there is less distinction made between the end-user and the programmer, and anyone who wants to start cranking out code has all the tools they need at their disposal.

Even if you're not a programmer, feel free to peruse the Kits as they appear to developers by using the Tracker to navigate the /boot/develop/headers/be hierarchy. Within that directory tree, you'll find a separate subdirectory for each of the Kits described in this chapter. All header files end with a .h extension. Open one of these files and it will be launched into the Metrowerks development environment, where BeOS programmers spend most of their late nights and early mornings crunching code to make your BeOS experience rock. See if you can determine the raison d'etre of a few random header files by studying the code they contain (of course, the filename is often a dead giveaway).

Since the Kits are constantly being updated and improved by the Be engineering team, it's possible—even probable—that the Kits outlined here may be different than the ones you find on your hard drive.

The Application Kit

The Application Kit is the most foundational of all the Kits—without it, you simply don't have a GUI application. The invocation of the Application Kit is the moment of conception for a BeOS application. By invoking the Application Kit, an application basically blinks into existence—it gains an identity and becomes known to the system. It gets registered in the Deskbar and can be referred to and communicated with by its application signature. The Application Kit consists of four major parts.

Note on terminology: A "class," in object-oriented programming terminology, is an object—a chunk of code that performs a specific task and that can be reused as often as necessary. Some classes are "public," and pertain to the entire operating system, while other classes are "private" and concern only the current application. The terms "class" and "object" are pretty much interchangeable.

Messaging The Application Kit allows programs to spawn threads (see Glossary or Chapter 2, *Meet the System*), and those threads to become the bearers of BMessages, enabling them to communicate with one another and with the threads of other running applications. It also handles "event messages," keeping track of mouse clicks, keypresses, and infrared signals from your garage door opener.

The BApplication Class By creating (or, more correctly, "instantiating") a BApplication class, the Application Kit makes its connection with the application server, which takes over common tasks like monitoring system resources, drawing images in windows, resizing borders, and lording over the cursor. Every BeOS application must include a single instance of this class.

The BRoster Class BeOS maintains at all times an internal roster of all running applications. Programmers can query this roster to learn which applications are currently available for communication. By registering themselves

Deeper than This

If this description of the BeOS system of Kits and servers makes the whole arrangement seem delightfully clean and simple, that's because it is … in concept. Compared to other operating systems, the BeOS API is a marvel of simplicity and elegance, and requires programmers to jump through far fewer hoops to get the system to do what they want it to do. It's so clean, in fact, that many BeOS developers have described programming for BeOS as "a joy." At the same time, though, the youth of the system means that there aren't as many prefab API calls in BeOS as there are in other operating systems—BeOS programmers sometimes have to install some of their own plumbing.

That fact notwithstanding, an operating system—even a brand spanking new one—is incredibly complex by definition. Each of the Kits described here is a small universe unto itself, and includes dozens or hundreds of objects, options, flags, arguments, and possibilities. The Kits are the thread and cloth from which every part of the operating system is woven, as well as the genetic map from which of all of your applications are built.

Just how complex are the Kits? Consider: If this book can be considered the bible for BeOS users, then the bible for BeOS programmers is *The Be Developer's Guide*, a book that covers the Kits and nothing but the Kits, in mind-numbing depth. The first edition of *The Be Developer's Guide* is more than 900 pages long. This chapter is much indebted to that book.

in the system roster, applications provide the data other programs need to establish communications.

The BClipboard Class This class handles the transfer of data to and from the system clipboard any time you cut, copy, or paste text, images, spreadsheet rows, or anything else.

The Storage Kit

We took a hard look at BFS (the Be Filesystem) in Chapter 5, *Files and the Tracker*, and discussed its many unique properties: journaling, attributes, node monitors, symbolic links, virtual filesystems, queries, and more. BFS is fine-tuned to handle high-bandwidth multimedia files with maximum throughput while at the same time offering the average user database-like functionality not available on any other system. The Storage Kit is the mechanism through which programmers instruct their applications to take advantage of these features. Of course, it's also responsible for the more mundane act of simply reading files from and to disk volumes. Needless to say, the Storage Kit works closely with the storage server to create and recognize files, directories, links, and attributes. Like the Application Kit, the Storage Kit is divided into four major functions.

Maintaining the Filesystem and Navigating Hierarchies While you can mount any number of disk drives on your system, each of these drives can contain multiple partitions, and these partitions can in turn support various filesystems, the totality of all mounted partitions appears as a single logical hierarchy when viewed by the Tracker, or when referred to from within programs. As you add or subtract partitions, the hierarchy maintains its integrity. The collection of all mounted partitions on your machine is referred to as "the filesystem." Note, however, that this term is a little confusing, since the term "filesystem" also applies to the low-level layout and formatted structure of a disk platter. One term, two meanings. Maintaining and communicating with the collective filesystem is the chief job of the Storage Kit.

Because BeOS uses a plug-in architecture that allows it to recognize filesystems from other operating systems, it must make at least one fundamental assumption about how those filesystems will be organized: BFS requires that any recognized filesystem will be hierarchical in nature, using a tree-like structure that fans out from a single root into a system of boughs, branches, and leaves (folders and files). Since virtually all filesystems out there are structured in this way, BeOS's potential support of alien filesystems is practically universal.

Under this umbrella fall dozens of related responsibilities.

Reading and Writing Files and Directories The function of the Storage Kit that is perhaps most visible to the user is the task of creating, deleting, naming, renaming, copying, moving, and otherwise managing the collection of mounted files and folders. This job is accomplished by way of two fundamental filesystem concepts: "nodes" and "entries." A node is an actual file—the data in your address book, for example. But it's not enough to simply scatter a disk with file data; something must keep track of what lives where so the filesystem can be constructed visually in the Tracker and reported accurately in the Terminal. Entries represent the locations of nodes. However, the relationship between nodes and entries is not necessarily one-to-one. While every node *must* have an entry, it is possible to have entries without nodes. For instance, an application might create an entry in the filesystem as a placeholder, before the file has actually been created.

As you might expect, the onus of maintaining MIME types and attributes (Chapter 5, *Files and the Tracker*) also falls on the shoulders of the Storage Kit. From the perspective of the BFS, attributes are essentially normal files, and have their own nodes and entries. But from the perspective of the Tracker and the Terminal, attributes are not displayed in file listings (by default, though you can of course customize any Tracker view to display an array of related attributes—see Chapter 2, *Meet the System*), nor are their sizes reported to the user, so a distinction must be made between them. Since the Storage Kit is responsible for all reading and writing of attributes and MIME types, the system FileTypes database (page 272 in Chapter 5) also falls under its rubric.

Monitoring Changes (Node Monitor) There are many instances in which it's useful for an application to be able to keep a watchful eye on a file or folder to see when it's been updated so it can take some kind of action. The mechanism that accomplishes this act of voyeurism is called the "node monitor." A simple example of the node monitor in action is the Trash icon on your desktop. As described in Chapter 5, the Trash is a special folder in a number of ways. One of its distinguishing characteristics is its ability to wear a different icon depending on whether it's empty or full. Because a node monitor is permanently trained on the Trash, dragging a file into the empty Trash bin will kick its node monitor into action, and the Trash icon will immediately change. The Tracker uses node monitors in all folder views—this is why making a change to the contents of a folder from the Terminal will cause the folder view in the GUI to be updated instantaneously. Application developers can employ node monitors in all kinds of creative ways. Node monitors are related to files and folders at the lowest level, so they're handled by the Storage Kit.

Asking for Files (Requests and Queries) Files don't do anyone any good if they can't be retrieved by applications. Even if it seems like the most trivial task, requesting a file from a disk volume is one of the Storage Kit's

most important jobs. There's more to this category than meets the eye, however. Requesting files from disk involves communicating with high-level system objects such as the familiar File panel and the Tracker list view, as well as more abstract functions, such as finding the current directory's parent, obtaining the full path of a file, or hoovering a file's custom icon out of its attributes.

The act of querying the system for files that meet certain criteria is another Storage Kit task falling under this umbrella. As intuitive as it may seem to our human brains, scanning the system for all files that begin with a capital "M" requires a substantial amount of work from BeOS. In addition, the Storage Kit needs to maintain queries in a "live" state so that the results window updates itself when files suddenly meet (or stop meeting) your criteria. Real-time updating of virtually everything is "the BeOS way" and is accomplished at the filesystem level by the labors of the Storage Kit.

The Storage Kit is also responsible for the job of creating and maintaining virtual filesystems such as /dev and /pipe (these directories are available to the system and can be seen from the Terminal, but do not represent actual storage locations on disk).

The Interface Kit

Because it's central to the construction and operation of almost all aspects of the BeOS GUI—especially for applications—the Interface Kit is the Big Kahuna: the largest and most complex of all the Kits. The Interface Kit works in close conjunction with the Application Kit to provide your applications' most commonly used objects. At the bare minimum, every application consists of a window and a "view" (views are containers inside windows, and play host to text spaces, image spaces, buttons and sliders, slider controls, oscilloscope readouts, backgrounds, etcetera). Providing windows, views, and all of the common interface elements you're accustomed to using in your applications is the job of the Interface Kit. This Kit also takes on all responsibilities related to drawing to the screen.

The relationship of views to windows is hierarchical, much like the filesystem. Whenever a window is created, it simultaneously creates a "top view" of exactly the same dimensions as the window, filling it completely. Everything filling the space of the application window exists as a "child" of the top view. The top view doesn't "do" anything, but acts as an umbrella for all the views it contains.

Every application window includes a top view, which acts as an umbrella object for everything else going on in the window. The children of the top view may or may not overlap one another within the window, and may or may not contain child views of their own. Figure 3 diagrams a hypothetical

application. The top (or parent) view is shown containing two major views (one where the user works and another that displays various status and reporting outputs representing the current state of that work). The work view contains a variety of control objects (slider, dial, text input, and button). The report view, in turn, is a parent to three more views, which will be laid out in the application space according to the developer's placement instructions. Boundaries between views may or may not be visible to the user. All child views inherit certain properties from their parents. For example, if the work view is defined as being 400 pixels wide, then text view, list view, and image view will inherit that 400-pixel "bounding box"—they won't be visible outside of that boundary no matter what the programmer does.

Drawing In order for anything to be situated inside a view, it must be drawn

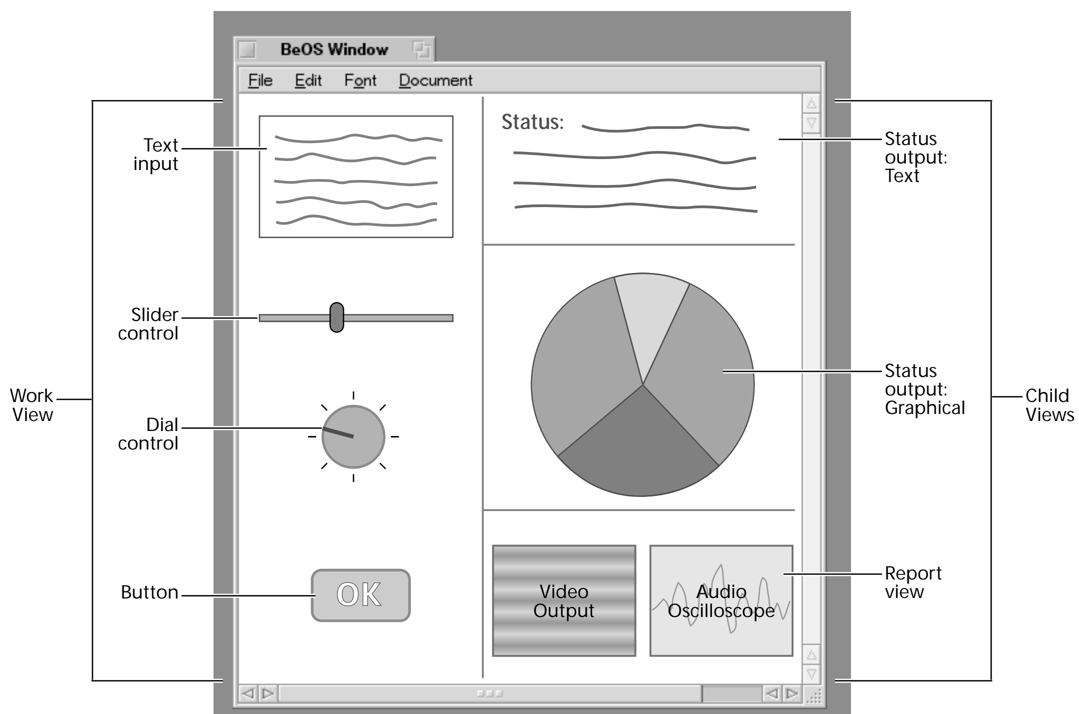


Figure 3

The window-to-view relationship in a hypothetical application. Note that the parent window hosts all of the views it contains, and that views can contain other views.

there. Drawing is the function of several subclasses of the Interface Kit, including the "pen," which establishes pixel coordinates and stroke widths, color management calls, drawing modes such as copy, erase, invert, transparency, and select, and an updating function that redraws screen areas to reflect changed information.

Interface Messages Another important responsibility of the Interface Kit is responding to the user's keyboard and mouse actions. When you press a letter on your keyboard, you want to see that letter onscreen immediately. When you double-click an application's title tab, you want that app to minimize. When you drag the "grippy dots" at the bottom right of a window, you want that window to be resized smoothly and instantly, the contents of its window reflowing if necessary. When you tap a menu trigger hotkey, you want to see that pulldown menu. All of these actions are known as "interface messages," and must be intercepted and responded to by code in the Interface Kit.

Character Encoding Meditate for a moment on what it would take to get BeOS working in French or German. Now imagine extending that translation job to Hebrew or Korean, where entirely different character sets are used. Obviously, this is a job much larger than simple translation—you can't simply pick a Hebrew font, hire a human translator, and expect everything to work like magic. Because of their complexity, foreign character sets require more bits of information per character than do languages using the standard ASCII set. ASCII encodes characters with eight bits, while complex character sets require 16 bits. The 16-bit encoding scheme agreed upon by international committee is called Unicode, and BeOS is fully Unicode aware.

While everything in the ASCII set can, of course, be encoded in the 16-bit system, the drawback to this is that you end up with a lot of wasted space when each character unnecessarily takes up 16 bits of memory. The solution to this dilemma is an encoding (which is sort of like a translation map) of Unicode called UTF-8 (UCS Transformation Format, where UCS stands for Universal Multiple-Octet Character Set, which is, in turn, simply longhand for Unicode). UTF-8 represents a multilingual encoding solution for BeOS by allowing complex character sets to be handled at the operating system level, while still retaining backward compatibility with the standard ASCII character set. Eight-bit encoding is actually somewhat more complex than that, but this is the gist of it. All Unicode encoding and decoding is handled by the Interface Kit.

Coordinate Spaces Determining where windows and objects are placed onscreen is another biggie for the Interface Kit. The system must know at all times where the top-left and bottom-right corners of every window and object are onscreen, and be able to place objects where the programmer wants them (initially) and where the user later moves them. This task is made somewhat more complex by the fact that users run their systems at different resolutions (Chapter 9, *Preferences*), and may opt to print at varying resolutions as well.

Text Views There are dozens of ways for text to manifest on your screen: in the main window of your word processor, in lists, on menu bars, on buttons,

in dialog box fields, and so on. Each of these has a separate API call defining individual characteristics, such as how many characters lines should wrap after, whether there's a limit on how much text can be entered into a space, whether the text is editable by the user, how to jump to a certain line, how to find text strings, whether it's OK to paste text into the view … all of this is handled by the Interface Kit.

Miscellany The more you study the BeOS interface, the more you realize how many different widgets are part of it, and how many considerations there are in accommodating applications no matter what they want to accomplish. Should a section of a dialog box be bounded with a fancy, raised border, or with a plain, "line" border? How should alert boxes look and behave? Should sliders have hash marks beneath them or not? Should the slider buttons be square, triangular, or some other shape? Should windows by resizable or not? How should your paint application's color picker appear? How should dialog boxes respond when the user changes the system font size? (This last question is not well-handled by BeOS R4, as you'll discover if you use system-wide fonts larger than a certain size—you'll notice that some of the text on buttons and in dialogs will be "clipped." Expect this to be fixed system-wide in a future release.) How can a program present a list of only monospaced fonts? How do developers get those little icons into your applications' toolbars? How should progress indicators appear, and how should they behave? How does the program bring a given view into focus, making it the target of the user's actions? The list goes on and on, and every little detail of every option related to everything in the user interface is made possible by the global code-base of the Interface Kit. It's a real doozy. In fact, the Interface Kit consumes the bulk of dead-tree mass in *The Be Developer's Guide*.

Kernel Kit

At the opposite end of the spectrum from the Interface Kit, which defines all the stuff the user sees and interacts with, is the Kernel Kit, which defines very low-level kernel services which are only experienced by the user indirectly.

Threads and Teams We've discussed BeOS's multithreadedness at several points in this book. Developers control threads and their communication with one another via the Kernel Kit. While threads function more or less as free agents (independently of one another), the group of threads that belong to a given application is referred to as a "team," and all of the threads in a team can work in conjunction with one another. Any time developers want to make their application more "fine-grained," they can spawn new threads to handle specific tasks. For instance, an Internet application that needs to check a Web site for updates might spawn a dedicated thread that does nothing but keep a watchful eye on a given URL, reporting its findings back to the application.

Because an application can't exist without threads, killing off a team (see Chapter 6, *The Terminal*, for thread- and team-killing techniques) has the result of killing the application.

It's worth pointing out that threads are very "inexpensive" on BeOS. That means that creating and destroying threads barely consumes any of the computer's resources at all. "When in doubt, spawn another thread" is a phrase that gets tossed around amongst developers occasionally. Just how cheap are threads? One real-time video demo created by Be engineers creates and destroys 60 threads *per second*. While few (if any) real-world applications call for that level of threading, it does demonstrate the fact that BeOS handles threads like nobody's business. If you see an application calling itself "heavily multithreaded," don't think for a moment that it means that the application is a resource hog!

Ports With all the messages moving around beneath the surface of your system at any given time, it becomes necessary to have some place for the threads to call home. A port is sort of like a mailbox for threads—they can deposit their messages in these virtual harbors so that other threads can come and pick them up for further processing. Ports are owned by the teams that create them. When the team dies or is killed off, they take their ports with them into the bit bucket.

Semaphores When studio musicians lay down independent tracks to be mixed together later, they need a way to synchronize their efforts. The most common way of doing this is via something called a "click track," a sort of metronome used to keep everyone working at the same tempo. The click track is used during the mixing stage to make sure all of the tracks are stored on the final recording in perfect synchronization.

Semaphores perform a similar function for threads, and are a sort of abstraction of the Kernel Kit. Any time a program needs to synchronize threads with one another, it uses a semaphore to "lock" the running function until the semaphore has been acquired. In other words, semaphores are sort of like guard dogs, blocking a thread's progress until it's safe to pass.

Areas As described in Chapter 9, *Preferences*, BeOS makes heavy use of virtual memory to cache parts of application code that are likely to be used again. Efficient use of virtual memory is one of the things that makes BeOS so fast. An "area" is a chunk of virtual memory roped off by the running application. If useful or necessary, an area can be made shareable so the data it contains is available to other applications. Alternatively, an area can be locked into RAM and optionally read/write protected.

Images To the Kernel Kit, images have nothing to do with graphics. Instead, they refer to chunks of compiled code that can be linked to a running applica-

tion. Every application has a single “app image,” while dynamically linked libraries have “library images,” and all of your add-ons have “add-on images.” Most developers don’t need to think about images; for most uses, the automatically created app image is all that’s needed.

Media Kit

The youngest member of the Kit family is the newly revamped Media Kit, which was almost completely rewritten from scratch for BeOS R4. The new Media Kit is a cornerstone in Be’s ongoing strategy to become the “SGI for the rest of us” and to make BeOS excel at creating and consuming professional digital content on consumer-level computers. The rewrite of the Media Kit also consolidated a number of services previously spread across various other Kits, such as the audio server (which continues to exist for backward-compatibility, but is officially “deprecated” by the new media server). The appearance of the new Media Kit also introduced a brand-new media server, which provides services to all types of timed media (with the primary emphasis being on audio and video in R4).

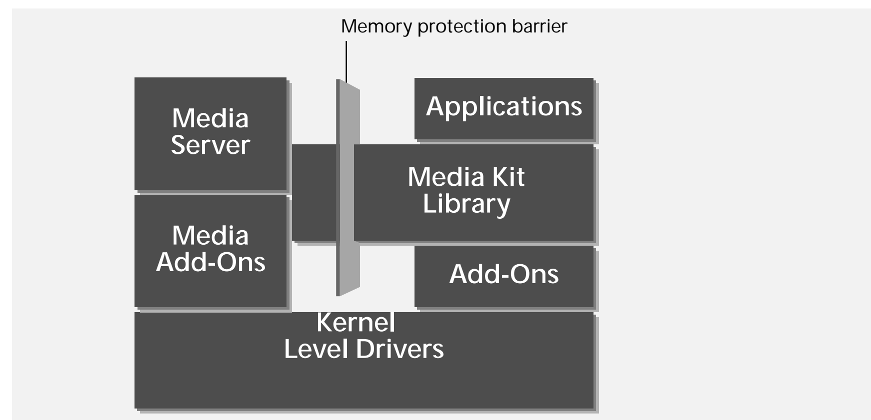


Figure 4

Like all of the Kits, the Media Kit is structured modularly. The application communicates with a media library and all of its add-ons, while the library communicates through the system’s memory protection barrier with the media server and media add-ons. The whole structure sits on top of the system’s kernel-level drivers (the driver for your video capture card, for example).

The Media Kit provides support for all forms of media, including the recording and playback of audio/visual signals, as well as access to a wide array of input/output media cards and devices. The Kit is structured around the concept of “nodes,” which are like media modules available to the system on which an application is running (you can think of a node as a sort of bridge between an application and a given driver).

The Media Kit's nodes are unrelated to the filesystem's nodes, described earlier. A node in the filesystem is an aread sketched out on the platter of your hard disk, while a media node is an abstract entity, existing in memory alone, and relating only to media functions.

In turn, there are two types of nodes: consumers and producers. Consumers accept media input and producers create it. Any node that is both a consumer and a producer can also be consider a "filter" because it can be used to alter the state of an audio or visual signal. BeOS maintains an internal roster (BMediaRoster) on each running system so that applications can easily discover which nodes are available to them for use. These nodes can be tied together into a virtual "web" of controls (also known as a "parameter web"), which is then accessible as a single entity from other applications.

One of the great things about this arrangement is that the developer has to do very little work to tap into BeOS's media capabilities. Rather than writing code to invoke an audio or video stream, a program just grabs onto a live node (or "instantiates" a dormant node from an installed add-on), connects it to a web, and requests a view in an application window to encapsulate whatever is moving through the web.

The new Media Kit also provides extremely tight timing functions that developers can use to synchronize audio soundtracks with video footage (for example). A program can optionally tell all of its nodes that they're "slaves" to a master timing signal, so that concurrently running audio and video streams stay in sync.

Because BeOS is so efficient, sound card latencies are far shorter than they are for the same sound card running under Windows or Linux. While latencies of up to 30 milliseconds are not uncommon under Windows, BeOS is capable of addressing sound cards at latencies of around six milliseconds. It's going to get even better than that, too. Be engineers are working toward a goal of one to two millisecond latencies, and early tests indicate that they should be able to reach this goal—on standard, consumer-level audio hardware no less. This efficiency, combined with the ease of development offered by the BeOS Media Kit, is seen as a great relief by many developers who have spent tedious hours trying to talk to audio hardware on other platforms at a low level.

Finally, the new Media Kit introduces a handful of codecs (compression/decompression algorithms—drivers that allow for the creation and translation of audio and video streams). As of R5, MIDI functions will also make their home in the Media Kit, rather than living in the previously separate MIDI Kit. The MIDI kit no longer exists a separate entity.

Network Kit

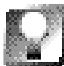
Obviously, the Network Kit provides all of the functions necessary to implement BeOS networking, whether over TCP/IP or UDP (UDP is a less common Internet protocol often used for Internet gaming or Webcast audio/video—it dispenses with error checking in favor of some speed gains). In addition, the Network Kit provides a mechanism by which add-ons can be written to enable BeOS support for common non-Internet protocols such as AppleTalk, NetBEUI, IPX, and others.

TCP/IP networking in BeOS is based on the open Berkeley Sockets (BSD) model. While the Network Kit once upon a time handled e-mail functions as well, that responsibility has been broken out into the separate Mail Kit. The Network Kit works in conjunction with the net_server to establish communications with various network protocols. It handles incoming and outgoing packets, IP addresses, and communication with network devices such as your PPP connection and/or network interface cards.

Game Kit

Any time you launch a game that takes over your entire screen, you know you're looking at the Game Kit in action. The Game Kit exists to offer maximum speed and direct access to underlying hardware. Since most games are hardware- and OS-intensive, this Kit provides developers with the means to maximize the overall quality of the user's gaming experience. While nothing in the Game Kit restricts its use to gaming, the BeBook coyly notes that even when the Kit is used for non-gaming purposes, "the user will have to deposit another 50 cents every three minutes."

Of course, one of the big benefits of the Game Kit is access to the BDirectWindow and BWindowScreen system calls, which allow applications to write massive amounts of video instructions straight to the guts of the video card, simultaneously bypassing most operating system overhead—an ideal state of affairs for intensive gaming action. BDirectWindow is also used for tasks like capturing or processing video streams, allowing tons of data to be written to the screen quickly. Read more about BDirectWindow on page 20 in Chapter 1, *The MediaOS*.



Remember: Any time an application has launched into Game Kit mode and doesn't provide an obvious escape hatch, you can quit by pressing Alt+Q or by toggling to another workspace and quitting the app from the Deskbar.

The Support Kit

While most of the Kits apply to specific categories of applications, the Support Kit is available to all applications, and contains a number of classes and utilities that provide extra functionality, such as the ability to turn applications into Replicants.

One of the most interesting aspects of the Support Kit is the BArchivable class, which allows objects to be copied into a static ("freeze-dried") form and handed off to another application, stored in memory cache, or saved out to a file. One of the most common implementations of the BArchivable class is to create Replicants (page 74 in Chapter 2, *Meet the System*)—application objects that are saved into a current state for indefinite hibernation (while your computer is off, for instance) and rehydrated later (when you reboot). As you know, Replicants can be embedded in other applications as well as in the Desktop; this is what "handing the object off to another application" means.

The Support Kit is also responsible for helping programmers to "lock" specific chunks of code, to make sure they remain safe from marauding bands of wild threads. All of the error codes presented to developers in the debugging process are stored in the Support Kit, as are various and sundry other items of interest only to programmers.

The Translation Kit

One of the really cool aspects of BeOS's object-oriented design is its use of "Translators" to negotiate between differing file formats, and to allow applications to open and save files and documents in any format for which a Translator is present. For example, a generic image-viewing application that starts life by supporting only GIF and JPEG images could be extended to also handle TIFF, TGA, PICT, BMP, and dozens of other formats with no intervention whatsoever from the author of the program, so long as the developer built Translator support into her application.

Similarly, a text editor could be extended to handle HTML, PostScript, or Word documents with the simple addition of a few new Translators to the appropriate folder (`/boot/home/add-ons/Translators`).

Translation isn't limited to open/save operations, however—its scope is more general than that. It can, for example, also be used to load objects directly into or out of system memory, or to read and write specific data formats to and from network connections.

To save developers the work of dealing with nitty-gritty details, all the major services of the Translation Kit are fed to the application by the translation roster, which queries the installed Translators as to their specific capabilities and

performs the specific implementation details of initialization, gathering information, running translation, and configuration.

Like other system add-ons in BeOS, Translators live in two separate directories—one at the system level and one at the user level:

```
/boot/beos/system/add-ons/translators
/boot/home/config/add-ons/translators
```

See Chapter 5, *Files and the Tracker*, for more on the user/system directory distinction.

System-level add-ons are preinstalled by BeOS and should not be tampered with, while you're free to add and remove Translators to and from the user-level directory. The two directories are read as a single unit by the system, and any Translators in the user directory that conflict with those in the system directory will override them.

The Mail Kit

Because email is such a critical component of our everyday computing lives, Be embeds mail services into the operating system much as it does with network services—another telling indication that BeOS is truly a modern operating system. Because of this, email services are not limited to dedicated email clients; for example, many programs allow you to send e-mail to the developer by clicking a button in the program's About box. Similarly, it would be possible for, say, a word processor to be extended to handle incoming and outgoing email messages.

There's an old saying in the Unix world that every application will grow until it's able to send and receive email. Fortunately, BeOS applications don't have to grow (in size) to acquire this capability—they can just open the door to the system's built-in services!

The Mail Kit allows for the configuration of POP and SMTP (incoming and outgoing) mail servers, scheduled mailbox checking, and the encoding and decoding of base-64 data used in some mail transfers. Each message created or stored via the Mail Kit is stored in the BeMail format, as a single file with an array of attributes for the sender, status, subject line, etcetera (see Chapter 4, *Get Online Fast*, for details). Crucial to the operation of the Mail Kit is the system `mail_daemon`, which sits in the background waiting to be kicked into action by calls from other programs (such as mail clients). The `mail_daemon`, in turn, makes sure that every email message file has the appropriate filetype (`text/x-email`) so that it will open in the preferred mail client when double-clicked.

Developers don't *have* to use the Mail Kit to make their programs send and receive email messages—they can wade through the standard piles of documentation and do it all by hand via the Network Kit if they want to. But if they're willing to have all messages stored in the BeMail format, they can use the Mail Kit as a shortcut, as Be has encapsulated all the nitty-gritty of message sending into a compact, ready-to-use package. This is especially a boon to developers who specialize in other areas. For example, if the developer of a paint and imaging application wants you to be able to send him support questions by clicking an icon in his program's toolbar, he shouldn't have to also learn the entire networking API just to email-enable his application—the Mail Kit makes it a piece of cake.

Chapter Summary

- The collection of Kits in BeOS is an exceptional example of many of the principles of object-orientation that developers have worked toward for years. BeOS's API isn't the first object-oriented API, but it is clean, logical, thoroughly modern, and makes programming for BeOS a pleasant experience in many ways. The layout of—and services provided by—the Kits means that developers can bring their software products to market faster than with other operating systems, and that those products will have less "cruft," or baggage, than is commonly carried around by similar programs on other operating systems.
- The object-oriented nature of the Kits mean that it's much simpler to extend applications into domains that would traditionally have required programmers to first sit down and read another stack of manuals. Because of the well-defined boundaries and interfaces provided by the Kits, extending an application to perform a new service is a little like snapping another LEGO onto an existing structure. I over-simplify here, but this is the general idea.
- All of this means an all-around better experience for users. It also means that if you're interested in learning to program in C++, BeOS is an excellent place to start, because the BeOS APIs are not overburdening. For more information on learning to program BeOS applications, see Appendix C.