1 Introduction

1 Introduction

The BeBox™ is an integrated package of hardware and software. The hardware supports the innovative design of the software, and the software exploits the extraordinary capabilities of the hardware. Among other things, the Be machine offers:

- Parallel processing on two high-performance CPUs.
- An operating system designed for efficient multithreading. It automatically splits assignments between the CPUs and will give priority to threads that need uninterrupted service.
- An architecture that supports the real-time processing of data for audio and video applications.
- An interface that lets applications and users view everything that's stored on-disk as if it were in a relational database.
- Dynamically loaded device drivers, built-in networking, interapplication messaging, shared libraries, protected and shared address spaces, an application framework that implicitly assigns a separate thread of execution to each window, and many other features.

Be system software is designed to make the features of the BeBox readily and efficiently available to all applications. The application programming interface (API) is written in the C++ language and takes advantage of the opportunities C++ offers for object-oriented programming. It includes numerous class definitions from which you can take much of the framework for your application.

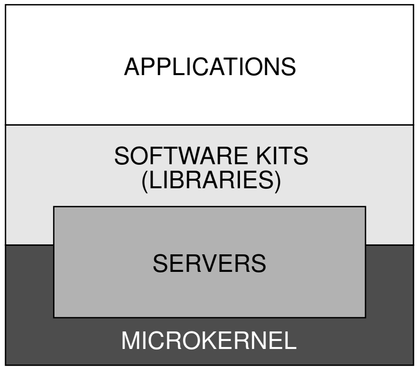
Software Overview

System software on the BeBox lies in three "layers":

- A microkernel that works directly with the hardware and device drivers.
- Several servers that can attend to the needs of any number of running applications. The servers take over much of the low-level work that would normally have to be done by each application.

- Dynamically-linked libraries that provide an interface to the servers and encapsulate facilities for building Be applications.

Applications are built on top of these layers, as illustrated below:



The application programming interface (API) for all system software is organized into several "kits." Each software kit has a distinct domain—there's a kit that contains the basic software you'll need to run an application on the BeBox, a kit for putting together a user interface, one for organizing data stored on-disk, another for networking, and so on.

With the exception of the Kernel and Network Kits, which have ANSI C interfaces, all the kits are written in the C++ language and make extensive use of class definitions. Each kit defines an integrated set of classes that work together to structure a framework for applications within its domain.

By incorporating kit classes in your application—directly creating instances of them, deriving your own classes from them, and inventing your own classes to work with them—you'll be able to make use of all the facilities built into the BeBox. And you'll find that a good deal of the work of programming a Be application has already been done for you by the engineers at Be.

Servers

Standing behind many of the software kits are servers—separate processes that run in the background and carry out basic tasks for client applications. Servers serve Be applications, not users; they have a programming interface (through the various kits) but no user interface. They typically can serve any number of running applications at the same time. A server can be viewed either as an extension of the kernel or as an adjunct to an application. It's really a little of both.

If you look inside the **/system** directory on the BeBox, you'll see a number of servers listed. The main ones that you should know about are the Storage Server and the Application Server.

- The *Storage Server* coordinates access to "persistent data"—data that lives on long-term storage media, such as a hard disk or floppy diskette. The Server keeps track of data by enumerating its qualities (its type, size, where it's located, and so on) in an entry called a *record*. In some cases, the record can hold the data itself; it can be a way of retaining data, not just of recording information about it. A *database* contains a collection of records; each medium (each hard or floppy disk) has its own database.

The Storage Server also manages the file system. A file, like any other distinct piece of persistent data, is represented by a record in a database. Although you gain access to files by referring to the records that represent them, Be software is designed to make file access as "database-free" as possible.

Your application can create new records, add them to a database, query a database and access records, open files to read and write, traverse directories, and carry out other storage and retrieval tasks through the classes defined in the Storage Kit. The Kit is the programming interface to the Server.

- The *Application Server* handles most of the low-level user interface work. It provides applications with windows, manages the interactions among windows, renders images in windows on instructions from the application, and monitors what the user does on the keyboard and mouse. It's the application's conduit for both drawing output and event input. In addition to being a "window provider," the Server also maintains the global environment shared by all applications.

  An application connects itself to the Server when it constructs a BApplication object (as defined in the Application Kit). This should be one of the first things that every application does. Every BWindow object (defined in the Interface Kit) also makes a connection to the Server when it's constructed. Each window runs independently of other windows—in its own thread and with its own connection to the Server.

Kits

Some of the software kits will be used by all applications, others only by applications that are concerned with the specific kinds of problems the kit addresses. Most applications will need to open files and put windows on-screen, for example; fewer will want to process audio data.

The kits currently available are summarized below:

- The *Application Kit* is a small amount of software that is nevertheless essential for all applications. It gives an application the ability to communicate with other applications, to become known to the Browser, and to use software in the other kits. It defines a messaging service that the system uses to report events to applications, and that an application can use to organize activity among its threads.

  The Kit's principal class is BApplication; every application must have one (and only one) BApplication object to act as its global representative. Begin with this kit before programming with any of the others.

- The *Storage Kit* is an interface for storing data on-disk, retrieving it, and keeping abreast of changes that are made to it. It's the client interface to the Storage Server. Information can be stored with various attached properties, so that it can be retrieved, accessed, and organized according to those properties, not just according to a file designation in a hierarchical directory structure.

  The Storage kit has two parts: One set of classes (BDatabase, BTable, BRecord, and BQuery) provides typical database access to stored information. Another set (BVolume, BDirectory, and BFile) provides an interface to the file system. The file-system classes are build on top of the database classes—files and directories have records in the database—but can be used with a minimum of "database" overhead. In the easiest (and most typical) case, an application doesn't need to know anything about database techniques to read and write files.

- The *Interface Kit* is used to build and run a graphical and interactive user interface. It structures the twin tasks of drawing in windows and handling the messages that report user actions (like clicks and keystrokes) directed at what was drawn. Its BWindow class encapsulates an interface to windows. Its BView class embodies a complete graphics environment for drawing.

  Each window on (and off) the screen is represented by a separate BWindow object and is served by a separate thread. A BWindow has a hierarchy of associated BView objects; each BView draws one portion of what's displayed in the window and responds to user actions prompted by the display. The Interface Kit defines a number of specific BViews, such as BListView, BButton, BScrollBar, and BTextView—as well as various supporting classes, such as BRegion, BBitmap, and BPicture.

  Every application that puts a window on-screen will need to make use of this kit.

- The *Media Kit* defines an architecture for the real-time processing of data—especially audio and video data. It gives applications the ability to generate, examine, manipulate, and realize (or "render") medium-specific data in real time. Applications can, for example, synchronize the transmission of data to different media devices, so they can easily incorporate and coordinate audio and video output.

- The *Midi Kit* is designed specifically for processing music data in MIDI (Musical Instrument Digital Interface) format.

- The *Kernel Kit* is the one kit that's not object-oriented. It defines an interface for creating threads (the basic units of scheduling and execution on the CPUs) and the attendant facilities that regulate threads and coordinate their interaction (such as ports, priorities, and semaphores). It also defines a system of memory management, including reserved and shared areas of memory. Applications that rely on the higher-level kits won't need to use much of the kernel interface.

- The *Device Kit* has two parts. One part provides programming interfaces to the various connectors on the BeBox; it currently consists of only the BSerialPort class. The other part is the API for creating loadable device drivers. Drivers for graphics cards run as extensions of the Application Server. All other drivers are loaded by the kernel.

- The *Network Kit* is for linking to and communicating with other computers through the TCP/IP and UDP/IP protocols. With only a couple of exceptions, its API is compatible with the BSD (Berkeley Software Distribution) UNIX socket architecture.

- The *Support Kit* is a collection of various defined types, error codes, and other facilities that support Be application development and the work of the other kits. It includes basic type definitions, the BList class for organizing ordered collections of data, and a system for having objects retain class information that they can reveal at run time. You can pick and choose the parts of this kit that you want to adopt for your application.

Contents

This manual documents system software for which a public API (application programming interface) is currently available. The present version covers eight kits—the Application, Storage, Interface, Media, Midi, Kernel, Device, and Support Kits summarized above. It doesn't yet cover the API for printing or for connecting to the telephone. Later releases will document more software as the API is codified.

After the introductory chapter you're now reading, there's a chapter for each kit. The table of contents is:

We may, from time to time, issue updated versions of one chapter or another, as well as add new chapters for new kits. So that page numbers won't become totally confusing as new documentation arrives, each chapter numbers its pages independently of the others. Each chapter begins on page 1 and has its own table of contents.

Where it can, the documentation tries to let you know what might be changing. It encloses temporary comments in angle brackets, <such as this>. Bracketed information is sometimes speculative, anticipating planned changes to the software that have yet to be implemented. Angle brackets sometimes also enclose information that's true about the present release, but is scheduled to change. Hopefully, language and context are enough to distinguish the two cases.

Just as the software tries to simplify the work of programming an application for the BeBox, this documentation tries to make it easy for you to understand the software. Your comments on it, as on the software, are appreciated. Suggestions, bug reports, and notes on what you found helpful or unhelpful, clear or unclear, are all welcome.

Class Descriptions

Since most Be software is organized into classes, much of the documentation you'll be reading in this manual will be about classes and their member functions. Each class description is divided into the following sections:

Overview — An introductory description of the class. The overview is usually brief, but for the main architectural classes, it can be lengthy. Start here to learn about the class.

Data Members — A list of the public and protected data members declared by the class, if there are any. If this section is missing, the class declares only private data members, or doesn't declare any data members at all. Most data members are private, so this section is usually absent.

Hook Functions — A list of the virtual functions that you're invited to override (re-implement) in a derived class. Hook functions are called by the kit at critical junctures; they "hook" application-specific code into the generic workings of the kit. Looking through the list will give you an idea of how to adapt the kit class to the needs of your application.

Constructor and Destructor — The class constructor and destructor. Only documented constructors produce valid members of a class. Don't rely on the default constructors promised by the C++ compiler.

Member Functions A full description of all public and protected member functions, including hook functions.

Operators A description of any operators that are overloaded to handle the class type.

If a section isn't relevant for a particular class—if the class doesn't define any hook functions or overload any operators, for example—that section is omitted.

Rely only on the documented API. You may occasionally find a public function declared in a header file but not documented in the class description. The reason it's not documented is probably because it's not supported and not safe; don't use it.

Programming Conventions

The software kits were designed with some conventions in mind. Knowing a few of these conventions will help you write efficient code and avoid unexpected pitfalls. The conventions for memory allocation, object creation, and virtual functions are described below.

Responsibility for Allocated Memory

The general rule is that whoever allocates memory is responsible for freeing it:

- If your application allocates memory, it should free it.
- If a kit allocates memory and passes your application a pointer to it, the kit retains responsibility for freeing it.

For example, a **Name()** function like this one,

```
char *name = someObject->Name();
```

would return a pointer to a string of characters residing in memory that belongs to the object that allocated it. The object will free the string; you shouldn't free it.

You should also not expect the string pointer to be valid for long. The object might modify the string, change its location in memory, reallocate it, or free it at any time. If your application needs continued access to the string, it should make a copy for itself or call **Name()** each time the string is needed.

In contrast, a **GetName()** function would copy the string into memory that your application provides:

```
char *name = (char *)malloc(someObject->NameLength() + 1);
someObject->GetName(name);
```

Your application is responsible for the copy.

In some cases, you're asked to allocate an object that kit functions fill in with data:

```
BPicture *picture = new BPicture;
someViewObject->BeginPicture(picture);
. . .
someViewObject->EndPicture();
```

Because your application allocated the object, it's responsible for freeing it.

Be system software tries always to keep allocation and deallocation paired in the same body of code—if you allocated the memory, free it; if you didn't, don't.

This general rule is followed wherever possible, but there are some exceptions to it. BMessage objects (in the Application Kit) are a prominent exception. Messages are like packages you put together and then mail to someone else. Although you create the package, once you mail it, it no longer belongs to you.

Another exception is **FindResource()** in the BResourceFile class of the Storage Kit. This function allocates memory on the caller's behalf and copies resource data to it; it then passes responsibility for the memory to the caller:

```
long numBytes;
void *res = someFile.FindResource(B_RAW_TYPE, "name", &numBytes);
```

The BResourceFile object allocates the memory in this case because it knows better than the caller how much resource data there is and, therefore, how much memory to allocate.

Exceptions like this are rare and are clearly stated in the documentation.

Object Allocation

All objects can be dynamically allocated (using the **new** operator). Some, but not all, can also be statically allocated (put on the stack). Static allocation is appropriate for certain kinds of objects, especially those that serve as temporary containers for transient data.

However, many objects may not work correctly unless they're allocated in dynamic memory. The general rule is this:

> *If you assign one object to another (as, for example, a child BView in the Interface Kit is assigned to its parent BView or a BMessage is assigned to a BMessenger), you should always dynamically allocate the assigned object.*

This is because there may be circumstances which would cause the other object to get rid of the object you assigned it. For example, a parent BView deletes its children when it is itself deleted. In the Be software kits, all such deletions are done with the **delete** operator. Therefore, the original allocation should always be done with **new**.

Virtual Functions

The software kits declare functions virtual for a variety of reasons. Most of the reasons simply boil down to this: Declaring a function virtual lets you reuse its name in a derived class. You can, for example, implement a special version of a function for one kind of object and give it the same name as the version defined in the kit for other objects. Or, if you always take certain steps when you call a particular kit function, you can reimplement the function to include those steps. You don't have to package your additions under a different name.

However, there's another, more important reason why some functions are declared virtual. These functions reverse the usual pattern for library functions: Instead of being implemented in the kit and called by the application, they're called by the kit and implemented in the application. They're "hooks" where you can hang your own code and introduce it into the on-going operations of the kit.

Hook functions are called at critical junctures as the application runs. They serve to notify the application that something has happened, or is about to happen, and they give the application a chance to respond.

For example, the BApplication class (in the Application Kit) declares a **ReadyToRun()** function that's called as the application is getting ready to run after being launched. It can be implemented to finishing configuring the application before it starts responding to the user. The BWindow class (in the Interface Kit) declares a **WindowActivated()** function that can be implemented to make any necessary changes when the window becomes the active window. By implementing these functions, you fit application-specific code into the generic framework of the kit.

It's possible to divide hook functions into three groups:

- Most hook functions are empty. As implemented by the declaring class, they don't do anything. It's up to derived classes to give them substance. Like **WindowActivated()** and **ReadyToRun()**, these functions are named for what they announce—for what led to the function call—rather than for what they might be implemented to do. They can be implemented to do almost anything you want.

- Some hook functions are given default implementations to cover the general case. Like the functions in the first group, these functions are also named for the occurrence that prompts the function call—for example, **ScreenChanged()** and **QuitRequested()**. If you decide to implement your own version of the function, you can choose either to *replace* the kit's default version or to *augment* it, as discussed below.

- A few hook functions are implemented to perform a particular task. You can call these functions just as you would any ordinary non-hook function, but they're also called at pivotal points within the framework of the kits. They therefore do double duty: They serve both as functions that you might call and as hooks that are called for you. These functions are generally named for what they do—like **MakeFocus()** or **SetValue()**. If you override any of them, you should always augment the original version, never replace it.

If you override a hook function that has been implemented—either by the class that declares it or by a derived class—it's generally best to preserve what the function already does by incorporating the old version in the new. For example:

```
void MyWindow::ScreenChanged(BRect grid, color_space mode)
{
    . . .
    BWindow::ScreenChanged(grid, mode);
    . . .
}
```

In this way, the new function augments the inherited version, rather than replaces it. It builds on what has already been implemented. In some cases, each class in a branch of the inheritance hierarchy will contribute a bit of code to a function. Because each version incorporates the inherited version, the function has its implementation spread vertically throughout the inheritance hierarchy.

Multiple Threads

A Be application is inherently multithreaded; it runs as a *team* of separately scheduled threads of execution that share a common address space. In addition to the *main thread* in which the application starts up and in which its **main()** function executes, each window is provided with its own thread. An application becomes multithreaded simply by creating a window.

Applications might explicitly create other threads for a variety of reasons—a thread might monitor a data channel, for example, or some less important processes might be put in a thread with a low priority to keep the user interface responsive. In addition, some kits (such as the Media Kit) have architectures that invite you to use multiple threads, and some spawn threads that work behind the scenes (like the thread that keeps live queries alive).

Each thread runs independently of the others, but the main thread has a special status. It's the first thread in the team, and it should also be the last. All other threads should be killed before the main thread and the application team are laid to rest.

The following sections discuss some considerations that come up when programming in a multithreaded environment. You may want to defer reading them until you see how the Be operating system defines and makes use of threads.

Protecting Data

Because all threads in a team live in the same address space, more than one of them might try to access the same data at the same time. If a data structure is static, unchanging, and certain to remain in place until the application quits, this won't be much of a problem. But that's generally not the case. If it's possible for one thread to alter some shared data, or delete it, while another thread is reading it (or worse, while it's also altering the data, but in a different way), obvious problems result. Data could be left in an internally inconsistent state, pointers could be invalid, and so on.

There are various way to avoid these problems—to keep critical data "multithread-safe". One maneuver is to put a single thread in charge of a data structure (or object). From the point of view of the data, the application isn't multithreaded; only one thread can read, alter, or delete it. Functions that deal with the data could simply return an error if the calling thread lacks authorized access.

The Be operating system provides two additional mechanisms that you can use to keep data multithread-safe:

- You can institute a locking procedure for the data. Locks are based on semaphores, which the Kernel Kit provides. Threads, in effect, wait in line to acquire a semaphore that gives them permission to access the data. When one thread releases the semaphore, the next thread can acquire it.

  Classes in some kits define **Lock()** and **Unlock()** functions that utilize semaphores, so it makes sense to talk about "locking" and "unlocking" an object—a window, for example.

  As long as all parties abide by the rules, semaphores and locks guarantee that only one thread at a time will be admitted to the data. Without this mechanism, one thread could not safely access data controlled by another thread.

- You can use the high-level messaging system, which the Application Kit defines. Messages asynchronously transfer control from one thread to another. They can make sure that a particular thread deals with particular data. For example, instead of locking an object and directly modifying its state, you can post a message to the thread that's associated with the object, and have that thread make the modification. If window *A* posts a message that concerns window *B*, window *B* will receive and respond to the message in its own thread.

  Using messages to communicate between threads ensures that each thread operates on just its "own" objects. For example, a window thread might accept messages that affect the window data structure (really a BWindow object) and other objects associated with the window. As long as other threads post messages rather than try for direct access, these objects will be accessed only from one thread.

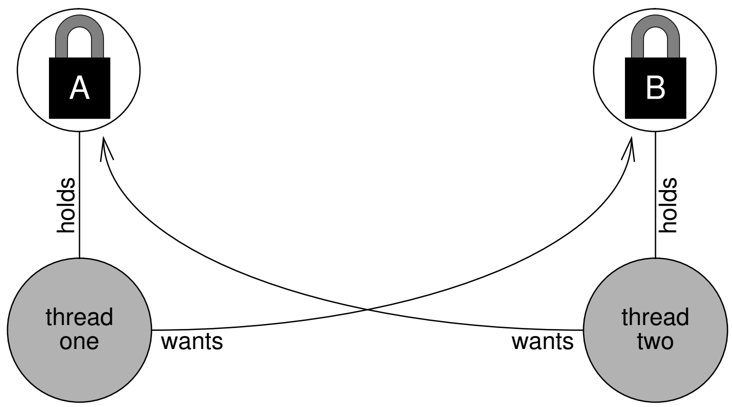
In the Be operating system, locks and messages are bound together in one important respect: When a thread receives a message, it automatically locks the object associated with the thread. For example, when a window thread gets a message, it locks the window data structure (the BWindow object). The lock remains in place until the thread is finished responding to the message.

This makes it possible for locks and messages to be used in combination in a multithreaded world. The choice of which to use depends on the situation and the design of your application.

The locking and messaging mechanisms are themselves multithread-safe on the BeBox. The system handles all the tricky cases—such as a destination thread disappearing while a message is being posted to it or a data structure being deleted while it's being locked. The functions that acquire a semaphore or lock and those that post messages are designed to fail gracefully and return an error if the objects of their attention have been destroyed.

Avoiding Deadlocks

A deadlock occurs when one thread tries to acquire a lock that another thread holds, while the other thread tries to acquire a lock that the first thread holds. This is diagrammed below:



Each thread blocks waiting to acquire the lock that the other thread holds. Neither will succeed because neither will release the lock it holds while it waits for the other thread to release its lock. They both wait forever—a deadlock. (Deadlocks can also involve a combination of three or more threads, but two are sufficient. The essential ingredient is that each thread waits for a lock that another thread holds while holding a lock of its own.)

As the diagram above indicates, there are two necessary conditions for a deadlock to occur. A deadlock can't happen unless:

- A thread that holds a lock tries for another one. If threads hold only one lock at a time, deadlocks can't occur.
- Two or more threads must try to acquire the same locks, but in a different sequence. In the illustration, thread one acquires lock *A* first, then tries for *B*, while thread two works in the opposite order. It acquires *B* first, then tries for *A*. If all threads always acquire locks in the same order, deadlocks can't occur.

If you structure your code to avoid either or both of these conditions, you won't experience deadlocks.

As mentioned earlier, when a thread receives a message it locks the object that owns the thread. Therefore, as a thread responds to a message it implicitly holds one lock. If it tries for another one, it will meet the first condition for a deadlock stated above.

At times this may be unavoidable. When it is, it's important to structure the code so that all threads try for the locks in the same order. For example, if window *X* and window *Y* need to share data, and window *X* can lock window *Y* and window *Y* can lock window *X*, there's a distinct possibility that a deadlock will sometime occur. If the information that each window needs from the other is moved to some third object under the supervision of another lock, a deadlock could be avoided. If more than one additional object is needed and more than one lock, both windows could acquire the external locks in the same order, avoiding a deadlock.

Sometimes the solution to a deadlock is to avoid locking and rely on messages instead. The two window in the example above might send each other messages rather than use locks to access the data directly.

Naming Conventions

As Be continues to develop system software and the API grows, there's a chance that the names of some new classes, constants, types, or functions added in future releases will clash with names you're already using in the code you've written.

To minimize the possibility of such clashes, we've adopted some strict naming conventions that will guide all future additions to the Be API. By stating these conventions here, we hope to give you a way of avoiding namespace conflicts in the future.

Most Be data structures and functions are defined as members of C++ classes, so class names will be quite prominent in application code. All our class names begin with the prefix "B"; the prefix marks the class as one that Be provides. The rest of the name is in mixed case—the body of the name is lowercase, but an uppercase letter marks the beginning of each separate word that's joined to form the name. For example:

BTextView	BFile
BRecord	BMessageQueue
BScrollBar	BList
BAudioSubscriber	BDatabase

The simplest thing you can do to prevent namespace clashes is to refrain from putting the "B" prefix on names you invent. Choose another prefix for your own classes, or use no prefix at all.

Other names associated with a class—the names of data members and member functions—are also in mixed case. (The names of member functions begin with an uppercase letter—for example, **AddResource()** and **UpdateIfNeeded()**. The names of data members begin with a lowercase letter—**what** and **bottom**, for example.) Member names are in a protected namespace and won't clash with the names you assign in your own code; they therefore don't have—or need—a "B" prefix.

All other names in the Be API are single case—either all uppercase or all lowercase—and use underbars to mark where separate words are joined into a single name.

The names of constants are all uppercase and begin with the prefix "B_". For example:

B_NAME_NOT_FOUND	**B_BACKSPACE**
B_OP_OVER	**B_LONG_TYPE**
B_BAD_THREAD_ID	**B_FOLLOW_TOP_BOTTOM**
B_REAL_TIME_PRIORITY	**B_PULSE**

It doesn't matter whether the constant is defined by a preprocessor directive (**#define**), in an enumeration (**enum**), or with the **const** qualifier. They're all uniformly uppercase, and all have a prefix. The only exceptions are common constants not specific to the Be operating system. For example, these four don't have a "B_" prefix:

TRUE	**NIL**
FALSE	**NULL**

Other names of whatever stripe—global variables, macros, nonmember functions, members of structures, and defined types—are all lowercase. Global variables generally begin with "be_",

be_app
be_roster
be_clipboard

but other names lack a prefix. They're distinguished only by being lowercase. For example:

rgb_color	**pattern**
system_time()	**acquire_sem()**
does_ref_conform()	**bytes_per_row**
app_info	**get_screen_size()**

There are few such names in the API. The software will grow mainly by adding classes and member functions, and the necessary constants to support those functions.

To briefly summarize:

Category	Prefix	Spelling
Class names	B	Mixed case
Member functions	*none*	Mixed case, beginning with an uppercase letter
Data members	*none*	Mixed case, beginning with a lowercase letter
Constants	**B_**	All uppercase
Global variables	**be_**	All lowercase
Everything else	*none*	All lowercase

If you adopt other conventions for your own code—perhaps mixed-case names, or possibly a prefix other than "B"—your names shouldn't conflict with any new ones we add in the future.

In addition, you can rely on our continuing to follow the lexical conventions established in the current API. For example, we never abbreviate “point” or “message,” but always abbreviate “rectangle” as “rect” and “information” as “info.” We use “begin” and “end,” never “start” or “finish,” in function names, and so on.

Occasionally, private names are visible in public header files. These names are marked with both pre- and postfixed underbars—for example, **_entry_** and **_remove_volume_()**. Don’t rely on these names in the code you write. They’re neither documented nor supported, and may change or disappear without comment in the next release.

Pre- and postfixed underbars are also used for kit-internal names that may intrude on an application’s namespace, even though they don’t show up in a header file. For example, the kits use some behind-the-scenes threads and give them names like “_pulse_task_” and they may put kit-internal data in public messages under names like “_button_”. If you were to assign the same names to your threads and data entries, they might conflict with kit code. Since you can’t anticipate every name used internally by the kits, it’s best to avoid all names that begin and end in underbars.