Kernel Extension Overview

As discussed in the chapter Kernel Architecture Overview, OS X provides a kernel extension mechanism as a means of allowing dynamic loading of code into the kernel, without the need to recompile or relink. Because these kernel extensions (KEXTs) provide both modularity and dynamic loadability, they are a natural choice for any relatively self-contained service that requires access to internal kernel interfaces.

Because KEXTs run in supervisor mode in the kernel's address space, they are also harder to write and debug than user-level modules, and must conform to strict guidelines. Further, kernel resources are wired (permanently resident in memory) and are thus more costly to use than resources in a user-space task of equivalent functionality.

In addition, although memory protection keeps applications from crashing the system, no such safeguards are in place inside the kernel. A badly behaved kernel extension in OS X can cause as much trouble as a badly behaved application or extension could in Mac OS 9.

Bugs in KEXTs can have far more severe consequences than bugs in user-level code. For example, a memory access error in a user application can, at worst, cause that application to crash. In contrast, a memory access error in a KEXT causes a kernel *panic*, crashing the operating system.

Finally, for security reasons, some customers restrict or don't permit the use of third-party KEXTs. As a result, use of KEXTs is strongly discouraged in situations where user-level solutions are feasible. OS X guarantees that threading in applications is just as efficient as threading inside the kernel, so efficiency should not be an issue. Unless your application requires low-level access to kernel interfaces, you should use a higher level of abstraction when developing code for OS X.

When you are trying to determine if a piece of code should be a KEXT, the default answer is generally *no*. Even if your code was a system extension in Mac OS 9, that does not necessarily mean that it should be a kernel extension in OS X. There are only a few good reasons for a developer to write a kernel extension:

- Your code needs to take a primary interrupt—that is, something in the (built-in) hardware needs to interrupt the CPU and execute a handler.
- The primary client of your code is inside the kernel—for example, a block device whose primary client is a file system.
- Your code needs to access kernel interfaces that are not exported to user space.
- Your code has other special requirements that cannot be satisfied in a user space application.

If your code does not meet any of the above criteria (and possibly even if it does), you should consider developing it as a library or a user-level daemon, or using one of the user-level plug-in architectures (such as QuickTime components or the Core Graphics framework) instead of writing a kernel extension.

If you are writing device drivers or code to support a new volume format or networking protocol, however, KEXTs may be the only feasible solution. Fortunately, while KEXTs may be more difficult to write than user-space code, several tools and procedures are available to enhance the development and debugging process. See Debugging Your KEXT for more information.

This chapter provides a conceptual overview of KEXTs and how to create them. If you are interested in building a simple KEXT, see the Apple tutorials listed in the bibliography. These provide step-by-step instructions for creating a simple, generic KEXT or a basic I/O Kit driver.

Implementation of a Kernel Extension (KEXT)

Kernel extensions are implemented as *bundles*, folders that the Finder treats as single files. See the chapter about bundles in *Mac Technology Overview* for a discussion of bundles.The KEXT bundle can contain the following:

- *Information property list*—a text file that describes the contents, settings, and requirements of the KEXT. This file is required. A KEXT bundle need contain nothing more than this file, although most KEXTs contain one or more kernel modules as well. See the chapter about software configuration in *Mac Technology Overview* for further information about property lists.
- *KEXT binary*—a file in Mach-O format, containing the actual binary code used by the KEXT. A KEXT binary (also known as a kernel module or *KMOD*) represents the minimum unit of code that can be loaded into the kernel. A KEXT usually contains one KEXT binary. If no KEXT binaries are included, the information property list file must contain a reference to another KEXT and change its default settings.
- *Resources*—for example, icons or localization dictionaries. Resources are optional; they may be useful for a KEXT that needs to display a dialog or menu. At present, no resources are explicitly defined for use with KEXTs.
- *KEXT bundles*—a kext can contain other KEXTs. This can be used for plug-ins that augment features of a KEXT.

Kernel Extension Dependencies

Any KEXT can declare that it is dependent upon any other KEXT. The developer lists these dependencies in the `OSBundleLibraries` dictionary in the module's property list file.

Before a KEXT is loaded, all of its requirements are checked. Those required extensions (and their requirements) are loaded first, iterating back through the lists until there are no more required extensions to load. Only after all requirements are met, is the requested KEXT loaded as well.

For example, device drivers (a type of KEXT) are dependent upon (require) certain families (another type of KEXT). When a driver is loaded, its required families are also loaded to provide necessary, common functionality. To ensure that all requirements are met, each device driver should list all of its requirements (families and other drivers) in its property list. See the chapter I/O Kit Overview, for an explanation of drivers and families.

It is important to list all dependencies for each KEXT. If your KEXT fails to do so, your KEXT may not load due to unrecognized symbols, thus rendering the KEXT useless. Dependencies in KEXTs can be considered analogous to required header files or libraries in code development; in fact, the Kernel Extension Manager uses the standard linker to resolve KEXT requirements.

Building and Testing Your Extension

After creating the necessary property list and C or C++ source files, you use Project Builder to build your KEXT. Any errors in the source code are brought to your attention during the build and you are given the chance to edit your source files and try again.

To test your KEXT, however, you need to leave Project Builder and work in the Terminal application (or in *console* mode). In console mode, all system messages are written directly to your screen, as well as to a log file (`/var/log/system.log`). If you work in the Terminal application, you must view system messages in the log file or in the Console application.You also need to log in to the *root* account (or use the `su` or `sudo` command), since only the root account can load kernel extensions.

When testing your KEXT, you can load and unload it manually, as well as check the load status. You can use the `kextload` command to load any KEXT. A manual page for `kextload` is included in OS X. (On OS X prior to 10.2, you must use the `kmodload` command instead.)

Note that this command is useful only when developing a KEXT. Eventually, after it has been tested and debugged, you install your KEXT in one of the standard places (see Installed KEXTs for details). Then, it will be loaded and unloaded automatically at system startup and shutdown or whenever it is needed (such as when a new device is detected).

Debugging Your KEXT

KEXT debugging can be complicated. Before you can debug a KEXT, you must first enable kernel debugging, as OS X is not normally configured to permit debugging the kernel. Only the root account can enable kernel debugging, and you need to reboot OS X for the changes to take effect. (You can use `sudo` to gain root privileges if you don't want to enable a root password.)

Kernel debugging is performed using two OS X computers, called the development or debug host and the debug target. These computers must be connected over a reliable network connection on the same subnet (or within a single local network). Specifically, there must not be any intervening IP routers or other devices that could make hardware-based Ethernet addressing impossible.

The KEXT is registered (and loaded and run) on the target. The debugger is launched and run on the debug host. You can also rebuild your KEXT on the debug host, after you fix any errors you find.

Debugging must be performed in this fashion because you must temporarily halt the kernel on the target in order to use the debugger. When you halt the kernel, all other processes on that computer stop. However, a debugger running remotely can continue to run and can continue to examine (or modify) the kernel on the target.

Note that bugs in KEXTs may cause the target kernel to freeze or panic. If this happens, you may not be able to continue debugging, even over a remote connection; you have to reboot the target and start over, setting a breakpoint just before the code where the KEXT crashed and working very carefully up to the crash point.

Developers generally debug KEXTs using *gdb*, a source-level debugger with a command-line interface. You will need to work in the Terminal application to run `gdb`. For detailed information about using `gdb`, see the documentation included with OS X. You can also use the `help` command from within `gdb`.

Some features of `gdb` are unavailable when debugging KEXTs because of implementation limitations. For example:

- You can't use `gdb` to call a function or method in a KEXT.
- You should not use `gdb` to debug interrupt routines.

The former is largely a barrier introduced by the C++ language. The latter may work in some cases but is not recommended due to the potential for `gdb` to interrupt something upon which *kdp* (the kernel shim used by `gdb`) depends in order to function properly.

Use care that you do not halt the kernel for too long when you are debugging (for example, when you set breakpoints). In a short time, internal inconsistencies can appear that cause the target kernel to panic or freeze, forcing you to reboot the target.

Additional information about debugging can be found in When Things Go Wrong: Debugging the Kernel.

Installed KEXTs

The Kernel Extension Manager (KEXT Manager) is responsible for loading and unloading all installed KEXTs (commands such as `kextload` are used only during development). Installed KEXTs are dynamically added to the running OS X kernel as part of the kernel's address space. An installed and enabled KEXT is invoked as needed.

Important: Note that KEXTs are only wrappers (bundles) around a property list, KEXT binaries (or references to other KEXTs), and optional resources. The KEXT describes what is to be loaded; it is the KEXT binaries that are actually loaded.

KEXTs are usually installed in the folder `/System/Libraries/Extensions`. The Kernel Extension Manager (in the form of a *daemon*, `kextd`), always checks here. KEXTs can also be installed in ROM or inside an application bundle.

Installing KEXTs in an application bundle allows an application to register those KEXTs without the need to install them permanently elsewhere within the system hierarchy. This may be more convenient and allows the KEXT to be associated with a specific, running application. When it starts, the application can register the KEXT and, if desired, unregister it on exit.

For example, a network packet sniffer application might employ a Network Kernel Extension (NKE). A tape backup application would require that a tape driver be loaded during the duration of the backup process. When the application exits, the kernel extension is no longer needed and can be unloaded.

Note that, although the application is responsible for registering the KEXT, this is no guarantee that the corresponding KEXTs are actually ever loaded. It is still up to a kernel component, such as the I/O Kit, to determine a need, such as matching a piece of hardware to a desired driver, thus causing the appropriate KEXTs (and their dependencies) to be loaded.