Building and Debugging Kernels

This chapter is not about building kernel extensions (KEXTs). There are a number of good KEXT tutorials on Apple's developer documentation site (http://developer.apple.com/documentation). This chapter is about adding new in-kernel modules (optional parts of the kernel), building kernels, and debugging kernel and kernel extension builds.

The discussion is divided into three sections. The first, Adding New Files or Modules, describes how to add new functionality into the kernel itself. You should only add files into the kernel when the use of a KEXT is not possible (for example, when adding certain low-level motherboard hardware support).

The second section, Building Your First Kernel, describes how to build a kernel, including how to build a kernel with debugger support, how to add new options, and how to obtain sources that are of similar vintage to those in a particular version of OS X or Darwin.

The third section, When Things Go Wrong: Debugging the Kernel, tells how to debug a kernel or kernel module using `ddb` and `gdb`. This is a must-read for anyone doing kernel development.

Adding New Files or Modules

In this context, the term module is used loosely to refer to a collection of related files in the kernel that are controlled by a single `config` option at compile time. It does not refer to loadable modules (KEXTs). This section describes how to add additional files that will be compiled into the kernel, including how to add a new `config` option for an additional module.

Modifying the Configuration Files

The details of adding a new file or module into the kernel differ according to what portion of the kernel contains the file. If you are adding a new file or module into the Mach portion of the kernel, you need to list it in various files in `xnu/osfmk/conf`. For the BSD portion of the kernel, you should list it in various files in `xnu/bsd/conf`. In either case, the procedure is basically the same, just in a different directory.

This section is divided into two subsections. The first describes adding the module itself and the second describes enabling the module.

Adding the Files or Modules

In the appropriate `conf` directory, you need to add your files or modules into various files. The files `MASTER`, `MASTER.ppc`, and `MASTER.i386` contain the list of configuration options that should be built into the kernel for all architectures, PowerPC, and i386, respectively.

These are supplemented by `files`, `files.ppc`, and `files.i386`, which contain associations between compile options and the files that are related to them for their respective architectures.

The format for these two files is relatively straightforward. If you are adding a new module, you should first choose a name for that module. For example, if your module is called `mach_foo`, you should then add a new option line near the top of `files` that is whitespace (space or tab) delimited and looks like this:

```
OPTIONS/mach_foo    optional mach_foo
```

The first part defines the name of the module as it will be used in `#if` statements in the code. (See Modifying the Source Code Files for more information.) The second part is always the word optional. The third part tells the name of the option as used to turn it on or off in a `MASTER` file. Any line with `mach_foo` in the last field will be enabled only if there is an appropriate line in a `MASTER` file.

Then, later in the file, you add

```
osfmk/foo/foo_main.c            optional mach_foo
osfmk/foo/foo_bar.c             optional mach_foo
```

and so on, for each new file associated with that module. This also applies if you are adding a file to an existing module. If you are adding a file that is not associated with any module at all, you add a line that looks like the following to specify that this file should always be included:

```
osfmk/crud/mandatory_file.c     standard
```

If you are not adding any modules, then you're done. Otherwise, you also need to enable your option in one of the `MASTER` files.

Enabling Module Options

To enable a module option (as described in the `files` files), you must add an entry for that option into one of the `MASTER` files. If your code is not a BSD pseudo-device, you should add something like the following:

```
options MACH_FOO
```

Otherwise, you should add something like this:

```
pseudo-device   mach_foo
```

In the case of a pseudo-device (for example, `/dev/random`), you can also add a number. When your code checks to see if it should be included, it can also check that number and allocate resources for more than one pseudo-device. The meaning of multiple pseudo-devices is device-dependent. An example of this is `ppp`, which allocates resources for two simultaneous PPP connections. Thus, in the `MASTER.ppc` file, it has the line:

```
pseudo-device   ppp 2
```

Modifying the Source Code Files

In the OS X kernel, all source code files are automatically compiled. It is the responsibility of the C file itself to determine whether its contents need to be included in the build or not.

In the example above, you created a module called `mach_foo`. Assume that you want this file to compile only on PowerPC-based computers. In that case, you should have included the option only in `MASTER.ppc` and not in `MASTER.i386`. However, by default, merely specifying the file `foo_main.c` in `files` causes it to be compiled, regardless of compile options specified.

To make the code compile only when the option mach_foo is included in the configuration, you should begin each C source file with the lines

```
#include <mach_foo.h>
#if (MACH_FOO > 0)
```

and end it with

```
#endif /* MACH_FOO */
```

If `mach_foo` is a pseudo-device and you need to check the number of `mach_foo` pseudo-devices included, you can do further tests of the value of `MACH_FOO`.

Note that the file `<mach_foo.h>` is not something you create. It is created by the makefiles themselves. You must run `make exporthdrs` before `make all` to generate these files.

Building Your First Kernel

Before you can build a kernel, you must first obtain source code. Source code for the OS X kernel can be found in the Darwin `xnu` project on http://www.opensource.apple.com. To find out your current kernel version, use the command `uname -a`. If you run into trouble, search the archives of the darwin-kernel and darwin-development mailing lists for information. If that doesn't help, ask for assistance on either list. The list archives and subscription information can be found at http://www.lists.apple.com.

Note: Before you begin, make sure you extract the sources in a directory whose path does not contain any "special" characters (non-alphanumeric characters other than dash and underscore), as having such characters in the path leading up to the build directory can cause compiling to fail.

Also, make sure that `/usr/local/bin` is in your `PATH` environment variable as follows:

If you are using a csh derivative such as tcsh, you should add `set path = (/usr/local/bin $path)` to your `.tcshrc` file

If you are using a Bourne shell derivative, you should add `export PATH=/usr/local/bin:$PATH` to your `.bashrc` file.

Important: Once you have obtained and extracted the sources, before you begin compiling kernel support tools, you should configure your system to build using gcc 3.3. The OS X v10.4 kernel will not build using gcc 4.0. To do this, type:

```
sudo gcc_select 3.3
```

Important: Before building anything, you should make sure you are running the latest version of OS X with the latest developer tools. The xnu compile process may reference various external headers from `/System/Library/Frameworks`. These headers are only installed as part of a developer tools installation, not as part of the normal OS X install process.

Next, you will need to compile several support tools. Get the `bootstrap_cmds`, `Libstreams`, `kext_tools`, `IOKitUser`, and `cctools` packages from http://www.opensource.apple.com. Extract the files from these `.tar` packages, then do the following:

```
sudo mkdir -p /usr/local/bin
sudo mkdir -p /usr/local/lib
cd bootstrap_cmds-version/relpath.tproj
make
sudo make install
```

```
cd ../../Libstreams-version
make
sudo make install
cd ../cctools-version
sudo cp /usr/include/ar.h \
        /System/Library/Frameworks/Kernel.framework/Headers
```

In the `cctools` package, modify the `Makefile`, and change the `COMMON_SUBDIRS` line (including the continuation line after it) to read:

```
COMMON_SUBDIRS = libstuff libmacho misc
```

Finally, issue the following commands:

```
make RC_OS=macos
sudo cp misc/seg_hack.NEW /usr/local/bin/seg_hack
cd ld
make RC_OS=macos kld_build
sudo cp static_kld/libkld.a /usr/local/lib
sudo ranlib /usr/local/lib/libkld.a
```

Now you're done with the cctools project. One final step remains: compiling `kextsymboltool`. To do this, extract the `kext_tools` tarball, then do the following:

```
sudo mkdir -p /System/Library/Frameworks/IOKit.framework/Versions/A/PrivateHeaders/kext
cd /System/Library/Frameworks/IOKit.framework/
sudo ln -s Versions/A/PrivateHeaders PrivateHeaders
sudo cp PATH_TO_IOKITUSER/IOKitUser-version/kext.subproj/*.h PrivateHeaders/kext
cd PATH_TO_KEXT_TOOLS/kext_tools-version
gcc kextsymboltool.c -o kextsymboltool
sudo cp kextsymboltool /usr/local/bin
```

> ⚠ **Warning:** If you do not use a version of `kextsymboltool` that is at least as current as your kernel, you will get serious compile failures. If you see the error message "exported name not in import list", there's a good chance you aren't using a current `kextsymboltool`.

Congratulations. You now have all the necessary tools, libraries, and header files to build a kernel.

The next step is to compile the kernel itself. First, change directories into the `xnu` directory. Next, you need to set a few environment variables appropriately. For your convenience, the kernel sources contain shell scripts to do this for you. If you are using sh, bash, zsh, or some other Bourne-compatible shell, issue the following command:

```
source SETUP/setup.sh
```

If you are using csh, tcsh, or a similar shell, use the following command:

```
source SETUP/setup.csh
```

Then, you should be able to type

```
make exporthdrs
make all
```

and get a working kernel in `BUILD/obj/RELEASE_PPC/mach_kernel` (assuming you are building a `RELEASE` kernel for PowerPC, of course).

If things don't work, the darwin-kernel mailing list a good place to get help.

Building an Alternate Kernel Configuration

When building a kernel, you may want to build a configuration other than the `RELEASE` configuration (the default shipping configuration). Additional configurations are `RELEASE_TRACE`, `DEBUG`, `DEBUG_TRACE`, and `PROFILE`. These configurations add various additional options (except `PROFILE`, which is reserved for future expansion, and currently maps onto `RELEASE`).

The most useful and interesting configurations are `RELEASE` and `DEBUG`. The release configuration should be the same as a stock Apple-released kernel, so this is interesting only if you are building source that differs from that which was used to build the kernel you are already running. Compiling a kernel without specifying a configuration results in the `RELEASE` configuration being built.

The `DEBUG` configuration enables `ddb`, the in-kernel serial debugger. The `ddb` debugger is helpful to debug panics that occur early in boot or within certain parts of the Ethernet driver. It is also useful for debugging low-level interrupt handler routines that cannot be debugged by using the more traditional `gdb`.

To compile an alternate kernel configuration, you should follow the same basic procedure as outlined previously, changing the final `make` statement slightly. For example, to build the `DEBUG` configuration, instead of typing

```
make all
```

you type

```
make KERNEL_CONFIGS=DEBUG all
```

and wait.

To turn on additional compile options, you must modify one of the `MASTER` files. For information on modifying these files, see the section Enabling Module Options.

When Things Go Wrong: Debugging the Kernel

No matter how careful your programming habits, sometimes things don't work right the first time. Kernel panics are simply a fact of life during development of kernel extensions or other in-kernel code.

There are a number of ways to track down problems in kernel code. In many cases, you can find the problem through careful use of `printf` or `IOLog` statements. Some people swear by this method, and indeed, given sufficient time and effort, any bug can be found and fixed without using a debugger.

Of course, the key words in that statement are "given sufficient time and effort." For the rest of us, there are debuggers: `gdb` and `ddb`.

Setting Debug Flags in Open Firmware

With the exception of kernel panics or calls to `PE_enter_debugger`, it is not possible to do remote kernel debugging without setting debug flags in Open Firmware. These flags are relevant to both `gdb` and `ddb` debugging and are important enough to warrant their own section.

To set these flags, you can either use the `nvram` program (from the OS X command line) or access your computer's Open Firmware. You can access Open Firmware this by holding down Command-Option-O-F at boot time. For most computers, the default is for Open Firmware to present a command-line prompt on your monitor and accept input from your keyboard. For some older computers you must use a serial line at 38400, 8N1. (Technically, such computers are not supported by OS X, but some are usable under Darwin, and thus they are mentioned here for completeness.)

From an Open Firmware prompt, you can set the flags with the `setenv` command. From the OS X command line, you would use the `nvram` command. Note that when modifying these flags you should always look at the old value for the appropriate Open Firmware variables and add the `debug` flags.

For example, if you want to set the debug flags to `0x4`, you use one of the following commands. For computers with recent versions of Open Firmware, you would type

```
printenv boot-args
setenv boot-args original_contents debug=0x4
```

from Open Firmware or

```
nvram boot-args
nvram boot-args="original_contents debug=0x4"
```

from the command line (as root).

For older firmware versions, the interesting variable is `boot-command`. Thus, you might do something like

```
printenv boot-command
setenv boot-command 0 bootr debug=0x4
```

from Open Firmware or

```
nvram boot-command
nvram boot-command="0 bootr debug=0x4"
```

from the command line (as root).

Of course, the more important issue is what value to choose for the debug flags. Table 20-1 lists the debugging flags that are supported in OS X.

Table 20-1 Debugging flags

Symbolic name	Flag	Meaning
`DB_HALT`	0x01	Halt at boot-time and wait for debugger attach (`gdb`).
`DB_PRT`	0x02	Send kernel debugging `printf` output to console.
`DB_NMI`	0x04	Drop into debugger on NMI (Command-Power, Command-Option-Control-Shift-Escape, or interrupt switch).

DB_KPRT	0x08	Send kernel debugging kprintf output to serial port.
DB_KDB	0x10	Make ddb (kdb) the default debugger (requires a custom kernel).
DB_SLOG	0x20	Output certain diagnostic info to the system log.
DB_ARP	0x40	Allow debugger to ARP and route (allows debugging across routers and removes the need for a permanent ARP entry, but is a potential security hole)—not available in all kernels.
DB_KDP_BP_DIS	0x80	Support old versions of gdb on newer systems.
DB_LOG_PI_SCRN	0x100	Disable graphical panic dialog.

The option DB_KDP_BP_DIS is not available on all systems, and should not be important if your target and host systems are running the same or similar versions of OS X with matching developer tools. The last option is only available in Mac OS 10.2 and later.

Avoiding Watchdog Timer Problems

Macintosh computers have various watchdog timers designed to protect the system from certain types of failures. There are two primary watchdog timers in common use: the power management watchdog timer (not present on all systems) and the system crash watchdog timer. Both watchdogs are part of the power management hardware.

The first of these, the power management watchdog timer, is designed to restore the system to a known safe state in the event of unexpected communication loss between the power management hardware and the CPU. This timer is *only* present in G4 and earlier desktops and laptops and in early G5 desktops. More specifically, it is present only in machines containing a PMU (Power Management Unit) chip.

Under normal circumstances, when communication with the PMU chip is lost, the PMU driver will attempt to get back in sync with the PMU chip. With the possible exception of a momentary loss of keyboard and mouse control, you probably won't notice that anything has happened (and you should never even experience such a stall unless you are writing a device driver that disables interrupts for an extended period of time).

The problem occurs when the disruption in communication is caused by entering the debugger while the PMU chip is in one of these "unsafe" states. If the chip is left in one of these "unsafe" states for too long, it will shut the computer down to prevent overheating or other problems.

This problem can be significantly reduced by operating the PMU chip in polled mode. This prevents the watchdog timer from activating. You should only use this option when debugging, however, as it diminishes performance and a crashed system could overheat.

To disable this watchdog timer, add the argument pmuflags=1 to the kernel's boot arguments. See Setting Debug Flags in Open Firmware for information about how to add a boot argument.

The second type of watchdog timer is the system crash watchdog timer. This is normally only enabled in OS X Server. If your target machine is running OS X Server, your system will automatically reboot within seconds after a crash to maximize server uptime. You can disable this automatic reboot on crash feature in the server administration tool.

Choosing a Debugger

There are two basic debugging environments supported by OS X: ddb and gdb. ddb is a built-in debugger that works over a serial line. By contrast, gdb is supported using a debugging shim built into

the kernel, which allows a remote computer on the same physical network to attach after a panic (or sooner if you pass certain options to the kernel).

For problems involving network extensions or low-level operating system bringups, `ddb` is the only way to do debugging. For other bugs, `gdb` is generally easier to use. For completeness, this chapter describes how to use both `ddb` and `gdb` to do basic debugging. Since `gdb` itself is well documented and is commonly used for application programming, this chapter assumes at least a passing knowledge of the basics of using `gdb` and focuses on the areas where remote (kernel) `gdb` differs.

Note: Only systems with serial hardware support `ddb`. Thus, it is only possible to use `ddb` on PowerMac G4 and older systems.

Using `gdb` for Kernel Debugging

`gdb`, short for the GNU Debugger, is a piece of software commonly used for debugging software on UNIX and Linux systems. This section assumes that you have used `gdb` before, and does not attempt to explain basic usage.

In standard OS X builds (and in your builds unless you compile with `ddb` support), `gdb` support is built into the system but is turned off except in the case of a kernel panic.

Of course, many software failures in the kernel do not result in a kernel panic but still cause aberrant behavior. For these reasons, you can pass additional flags to the kernel to allow you to attach to a remote computer early in boot or after a nonmaskable interrupt (NMI), or you can programmatically drop into the debugger in your code.

You can cause the test computer (the debug target) to drop into the debugger in the following ways:

- debug on panic
- debug on NMI
- debug on boot
- programmatically drop into the default debugger

  The function `PE_enter_debugger` can be called from anywhere in the kernel, although if `gdb` is your default debugger, a crash will result if the network hardware is not initialized or if `gdb` cannot be used in that particular context. This call is described in the header `pexpert/pexpert.h`.

After you have decided what method to use for dropping into the debugger on the target, you must configure your debug host (the computer that will actually be running `gdb`). Your debug host should be running a version of OS X that is comparable to the version running on your target host. However, it should not be running a customized kernel, since a debug host crash would be problematic, to say the least.

Note: It is possible to use a non-OS X system as your debug host. This is not a trivial exercise, however, and a description of building a cross-`gdb` is beyond the scope of this document.

When using `gdb`, the best results can be obtained when the source code for the customized kernel is present on your debug host. This not only makes debugging easier by allowing you to see the lines of code when you stop execution, it also makes it easier to modify those lines of code. Thus, the ideal situation is for your debug host to also be your build computer. This is not required, but it makes things easier. If you are debugging a kernel extension, it generally suffices to have the source for the kernel extension itself on your debug host. However, if you need to see kernel-specific structures, having the kernel sources on your debug host may also be helpful.

Once you have built a kernel using your debug host, you must then copy it to your target computer and reboot the target computer. At this point, if you are doing panic-only debugging, you should trigger the

panic. Otherwise, you should tell your target computer to drop into the debugger by issuing an NMI (or by merely booting, in the case of `debug=0x1`).

Next, unless your kernel supports ARP while debugging (and unless you enabled it with the appropriate debug flag), you need to add a permanent ARP entry for the target. It will be unable to answer ARP requests while waiting for the debugger. This ensures that your connection won't suddenly disappear. The following example assumes that your target is `target.foo.com` with an IP number of `10.0.0.69`:

```
$ ping -c 1 target_host_name
ping results: ....
$ arp -an
target.foo.com (10.0.0.69): 00:a0:13:12:65:31
$ sudo arp -s target.foo.com 00:a0:13:12:65:31
$ arp -an
target.foo.com (10.0.0.69) at00:a0:13:12:65:31 permanent
```

Now, you can begin debugging by doing the following:

```
gdb /path/to/mach_kernel
source /path/to/xnu/osfmk/.gdbinit
p proc0
source /path/to/xnu/osfmk/.gdbinit
target remote-kdp
attach 10.0.0.69
```

Note that the mach kernel passed as an argument to `gdb` should be the symbol-laden kernel file located in `BUILD/obj/DEBUG_PPC/mach_kernel.sys` (for debug kernel builds, `RELEASE_PPC` for non-debug builds), not the bootable kernel that you copied onto the debug target. Otherwise most of the `gdb` macros will fail. The correct kernel should be several times as large as a normal kernel.

You must do the `p proc0` command and source the `.gdbinit` file (from the appropriate kernel sources) twice to work around a bug in `gdb`. Of course, if you do not need any of the macros in `.gdbinit`, you can skip those two instructions. The macros are mostly of interest to people debugging aspects of Mach, though they also provide ways of obtaining information about currently loaded KEXTs.

> ⚠ **Warning:** It may not be possible to detach in a way that the target computer's kernel continues to run. If you detach, the target hangs until you reattach. It is not always possible to reattach, though the situation is improving in this area. Do not detach from the remote kernel!

If you are debugging a kernel module, you need to do some additional work to get debugging symbol information about the module. First, you need to know the load address for the module. You can get this information by running `kextstat` (`kmodstat` on systems running OS X v10.1 or earlier) as root on the target.

If you are already in the debugger, then assuming the target did not panic, you should be able to use the `continue` function in `gdb` to revive the target, get this information, then trigger another NMI to drop back into the debugger.

If the target is no longer functional, and if you have a fully symbol-laden kernel file on your debug host that matches the kernel on your debug target, you can use the `showallkmods` macro to obtain this information. Obtaining a fully symbol-laden kernel generally requires compiling the kernel yourself.

Once you have the load address of the module in question, you need to create a symbol file for the module. You do this in different ways on different versions of OS X.

For versions 10.1 and earlier, you use the `kmodsyms` program to create a symbol file for the module. If your KEXT is called `mykext` and it is loaded at address 0xf7a4000, for example, you change directories to `mykext.kext/Contents/MacOS` and type:

```
kmodsyms -k path/to/mach_kernel -o mykext.sym mykext@0xf7a4000
```

Be sure to specify the correct path for the mach kernel that is running on your target (assuming it is not the same as the kernel running on your debug host).

For versions after 10.1, you have two options. If your KEXT does not crash the computer when it loads, you can ask `kextload` to generate the symbols at load time by passing it the following options:

```
kextload -s symboldir mykext.kext
```

It will then write the symbols for your kernel extension and its dependencies into files within the directory you specified. Of course, this only works if your target doesn't crash at or shortly after load time.

Alternately, if you are debugging an existing panic, or if your KEXT can't be loaded without causing a panic, you can generate the debugging symbols on your debug host. You do this by typing:

```
kextload -n -s symboldir mykext.kext
```

If will then prompt you for the load address of the kernel extension and the addresses of all its dependencies. As mentioned previously, you can find the addresses with `kextstat` (or `kmodstat`) or by typing `showallkmods` inside `gdb`.

You should now have a file or files containing symbolic information that `gdb` can use to determine address-to-name mappings within the KEXT. To add the symbols from that KEXT, within `gdb` on your debug host, type the command

```
add-symbol-file mykext.sym
```

for each symbol file. You should now be able to see a human-readable representation of the addresses of functions, variables, and so on.

Special `gdb` I/O Addressing Issues

As described in Address Spaces, some Macintosh hardware has a third addressing mode called I/O addressing which differs from both physical and virtual addressing modes. Most developers will not need to know about these modes in any detail.

Where some developers may run into problems is debugging PCI device drivers and attempting to access device memory/registers.

To allow I/O-mapped memory dumping, do the following:

```
set kdp_read_io=1
```

To dump in physical mode, do the following:

```
set kdp_trans_off=1
```

For example:

```
(gdb) x/x 0xf8022034
```

```
0xf8022034: Cannot access memory at address 0xf8022034
(gdb) set kdp_trans_off=1
(gdb) x/x 0xf8022034
0xf8022034: Cannot access memory at address 0xf8022034
(gdb) set kdp_read_io=1
(gdb) x/x 0xf8022034
0xf8022034: 0x00000020
(gdb)
```

If you experience problems accessing I/O addresses that are not corrected by this procedure, please contact Apple Developer Technical Support for additional assistance.

Using ddb for Kernel Debugging

When doing typical debugging, gdb is probably the best solution. However, there are times when gdb cannot be used or where gdb can easily run into problems. Some of these include

- drivers for built-in Ethernet hardware
- interrupt handlers (the hardware variety, not handler threads)
- early bootstrap before the network hardware is initialized

When gdb is not practical (or if you're curious), there is a second debug mechanism that can be compiled into OS X. This mechanism is called ddb, and is similar to the kdb debugger in most BSD UNIX systems. It is not quite as easy to use as gdb, mainly because of the hardware needed to use it.

Unlike gdb (which uses Ethernet for communication with a kernel stub), ddb is built into the kernel itself, and interacts directly with the user over a serial line. Also unlike gdb, using ddb requires building a custom kernel using the DEBUG configuration. For more information on building this kernel, see Building Your First Kernel.

Note: ddb requires an actual *built-in hardware* serial line on the debug target. Neither PCI nor USB serial adapters will work. In order to work reliably for interrupt-level debugging, ddb controls the serial ports directly with a polled-mode driver without the use of the I/O Kit.

If your debug target does not have a factory serial port, third-party adapter boards may be available that replace your internal modem with a serial port. Since these devices use the built-in serial controller, they should work for ddb. It is not necessary to install OS X drivers for these devices if you are using them only to support ddb debugging.

The use of these serial port adapter cards is not an officially supported configuration, and not all computers support the third-party adapter boards needed for ddb support. Consult the appropriate adapter board vendor for compatibility information.

If your target computer has two serial ports, ddb uses the modem port (SCC port 0). However, if your target has only one serial port, that port is probably attached to port 1 of the SCC cell, which means that you have to change the default port if you want to use ddb. To use this port (SCC port 1), change the line:

```
const int console_unit=0;
```

in osfmk/ppc/serial_console.c to read:

```
const int console_unit=1;
```

and recompile the kernel.

Once you have a kernel with `ddb` support, it is relatively easy to use. First, you need to set up a terminal emulator program on your debug host. If your debug host is running Mac OS 9, you might use `ZTerm`, for example. For OS X computers, or for computers running Linux or UNIX, `minicom` provides a good environment. Setting up these programs is beyond the scope of this document.

Important: Serial port settings for communicating with `ddb` must be 57600 8N1. Hardware handshaking may be on, but is not necessary.

Note: For targets whose Open Firmware uses the serial ports, remember that the baud rate for communicating with Open Firmware is 38400 and that hardware handshaking must be *off*.

Once you boot a kernel with `ddb` support, a panic will allow you to drop into the debugger, as will a call to `PE_enter_debugger`. If the `DB_KDB` flag is not set, you will have to press the D key on the keyboard to use `ddb`. Alternately, if both `DB_KDB` and `DB_NMI` are set, you should be able to drop into `ddb` by generating a nonmaskable interrupt (NMI). See Setting Debug Flags in Open Firmware for more information on debug flags.

To generate a nonmaskable interrupt, hold down the command, option, control, and shift keys and hit escape (OS X v10.4 and newer), hold down the command key while pressing the power key on your keyboard (on hardware with a power key), or press the interrupt button on your target computer. At this point, the system should hang, and you should see `ddb` output on the serial terminal. If you do not, check your configuration and verify that you have specified the correct serial port on both computers.

Commands and Syntax of `ddb`

The `ddb` debugger is much more `gdb`-like than previous versions, but it still has a syntax that is very much its own (shared only with other `ddb` and `kdb` debuggers). Because `ddb` is substantially different from what most developers are used to using, this section outlines the basic commands and syntax.

The commands in `ddb` are generally in this form:

```
command[/switch] address[,count]
```

The switches can be one of those shown in Table 20-2.

Table 20-2 Switch options in `ddb`

Switch	Description
`/A`	Print the location with line number if possible
`/I`	Display as instruction with possible alternate machine-dependent format
`/a`	Print the location being displayed
`/b`	Display or process by bytes
`/c`	Display low 8 bits as a character (nonprinting characters as octal) *or* count instructions while executing (depends on instruction)
`/d`	Display as signed decimal
`/h`	Display or process by half word (16 bits)

`/i`	Display as an instruction
`/l`	Display or process by long word (32 bits)
`/m`	Display as unsigned hex with character dump for each line
`/o`	Display in unsigned octal
`/p`	Print cumulative instruction count and call tree depth at each call or return statement
`/r`	Display in current radix, signed
`/s`	Display the null-terminated string at address (nonprinting as octal).
`/u`	Display in unsigned decimal *or* set breakpoint at a user space address (depending on command).
`/x`	Display in unsigned hex
`/z`	Display in signed hex

The `ddb` debugger has a rich command set that has grown over its lifetime. Its command set is similar to that of `ddb` and `kdb` on other BSD systems, and their manual pages provide a fairly good reference for the various commands. The command set for `ddb` includes the following commands:

`break[/u] addr`
Set a breakpoint at the address specified by `addr`. Execution will stop when the breakpoint is reached. The `/u` switch means to set a breakpoint in user space.

`c` *or* `continue[/c]`
Continue execution after reaching a breakpoint. The `/c` switch means to count instructions while executing.

`call`
Call a function.

`cond`
Set condition breakpoints. This command is not supported on PowerPC.

`cpu cpunum`
Causes `ddb` to switch to run on a different CPU.

`d` *or* `delete [addr|#]`
Delete a breakpoint. This takes a single argument that can be either an address or a breakpoint number.

`dk`
Equivalent to running `kextstat` while the target computer is running. This lists loaded KEXTs, their load addresses, and various related information.

`dl vaddr`
Dumps a range of memory starting from the address given. The parameter `vaddr` is a kernel virtual address. If `vaddr` is not specified, the last accessed address is used. See also `dr`, `dv`.

`dm`
Displays mapping information for the last address accessed.

`dmacro name`
Delete the macro called `name`. See `macro`.

`dp`
Displays the currently active page table.

`dr addr`

Dumps a range of memory starting from the address given. The parameter `address` is a physical address. If `addr` is not specified, the last accessed address is used. See also `dl`, `dv`.

`ds`
Dumps save areas of all Mach tasks.

`dv [addr [vsid]]`
Dumps a range of memory starting from the address given. The parameter `addr` is a virtual address in the address space indicated by `vsid`. If `addr` is not specified, the last accessed address is used. Similarly, if `vsid` is not specified, the last `vsid` is used. See also `dl`, `dr`.

`dwatch addr`
Delete a watchpoint. See `watch`.

`dx`
Displays CPU registers.

`examine`
See `print`.

`gdb`
Switches to `gdb` mode, allowing `gdb` to attach to the computer.

`lt`
On PowerPC only: Dumps the PowerPC exception trace table.

`macro name command [ ; command .. ]`
Create a macro called `name` that executes the listed commands. You can show a macro with the command `show macro name` or delete it with `dmacro name`.

`match[/p]`
Stop at the matching return instruction. If the `/p` switch is not specified, summary information is printed only at the final return.

`print[/AIabcdhilmorsuxz] addr1 [addr2 ...]`
Print the values at the addresses given in the format specified by the switch. If no switch is given, the last used switch is assumed. Synonymous with `examine` and `x`. Note that some of the listed switches may work for `examine` and not for `print`.

`reboot`
Reboots the computer. Immediately. Without doing any file-system unmounts or other cleanup. Do not do this except after a panic.

`s` *or* `step`
Single step through instructions.

`search[/bhl] addr value [mask[,count]]`
Search memory for `value` starting at `addr`. If the value is not found, this command can wreak havoc. This command may take other formatting values in addition to those listed.

`set $name [=] expr`
Sets the value of the variable or register named by `name` to the value indicated by `expr`.

`show`
Display system data. For a list of information that can be shown, type the `show` command by itself. Some additional options are available for certain options, particularly `show all`. For those suboptions, type `show all` by itself.

`trace[/u]`
Prints a stack backtrace. If the `/u` flag is specified, the stack trace extends to user space if supported by architecture-dependent code.

`until[/p]`
Stop at the next call or return.

`w` *or* `write[/bhl] addr expr1 [expr2 ... ]`
Writes the value of `expr1` to the memory location stored at `addr` in increments of a byte, half word, or long word. If additional expressions are specified, they are written to consecutive bytes, half words, or long words.

`watch addr[,size]`

Sets a watchpoint on a particular address. Execution stops when the value stored at that address is modified. Watch points are not supported on PowerPC.

> ⚠ **Warning** Watching addresses in wired kernel memory may cause unrecoverable errors on i386.

`x`
Short for `examine`. See `print`.

`xb`
Examine backward. Execute the last examine command, but use the address previous to the last one used (jumping backward by increments of the last width displayed).

`xf`
Examine forward. Execute the last `examine` command, but use the address following the last one used (jumping by increments of the last width displayed).

The `ddb` debugger should seem relatively familiar to users of `gdb`, and its syntax was changed radically from its predecessor, `kdb`, to be more `gdb`-like. However, it is still sufficiently different that you should take some time to familiarize yourself with its use before attempting to debug something with it. It is far easier to use `ddb` on a system whose memory hasn't been scribbled upon by an errant DMA request, for example.
