Chapter 11

Testing and Debugging a Driver

As a critical system component, a driver deserves careful testing, but because it is part of the kernel, the normal testing tools are not available. This chapter describes the available testing tools and methods, in the following major topics:

- "Preparing the System for Debugging" on page 243 describes how to set up the kernel for use of the debugging tools.
- "Producing Diagnostic Displays" on page 249 covers the kernel functions your driver can use to generate diagnostic output as it executes.
- "Using symmon" on page 252 describes the use of the standalone debugger.
- "Using idbg" on page 262 describes some uses of the kernel-display command.

Preparing the System for Debugging

The standalone debugger *symmon* is a key tool for driver programming. It must be installed in the volume header of the boot disk. In order for it to be useful you must boot a "debugging" kernel, that is, one that retains symbols, and contains the display modules, that are used by debugging tools. Normally these modules and symbols are eliminated to save space. You modify the *irix.sm* file to enable debugging, and then generate a new kernel.

All these steps should be performed before you attempt to install your device driver.

Placing symmon in the Volume Header

The *symmon* standalone debugger resides in the volume header of a disk—not in a normal IRIX filesystem. The volume header is disk partition 0. It always contains a label record (sgilabel), and the standalone shell *sash* that manages the bootstrap operation. On some systems it may also contain the *ide* program, a PROM-level diagnostic program. If *symmon* is to be available, it, too, must be placed in the volume header.

Normally you acquire *symmon* by installing the debugging kernel feature (eoe.sw.kdebug) in the IRIX Developer Option software distribution. You can verify that this feature has been installed by executing the command

```
versions eoe.sw.kdebug
```

The response should confirm the presence of this component (it does not show *symmon* by name). When you install the kernel debug feature, the *symmon* program file is copied to the volume header of the current boot disk automatically.

You can verify the presence of *symmon* in the volume header through the use of *dvhtool* (described in the dvhtool(1) reference page). The results should be similar to the display in Example 11-1. The response to the "l" (list) command shows that the volume header of this disk contains *sgilabel*, *ide*, *sash*, and *symmon*.

Example 11-1 Verifying Presence of symmon

```
# dvhtool
Volume? (/dev/rvh) /dev/rvh
Command? (read, vd, pt, dp, write, bootfile, or quit): vd
(d FILE, a UNIX_FILE FILE, c UNIX_FILE FILE, g FILE UNIX_FILE or l)?
        l
Current contents:
        File name        Length     Block #
        sgilabel            512           2
        ide              281600         278
        sash             281600         828
        symmon           248320        1378
(d FILE, a UNIX_FILE FILE, c UNIX_FILE FILE, g FILE UNIX_FILE or l)?

Command? (read, vd, pt, dp, write, bootfile, or quit): quit
#
```

In the event you need to install *symmon* in the volume header of a disk without using the software manager, you can copy the standalone program to the volume header using *dvhtool*. However, you first need to get a copy of the program in the form of a UNIX file.

Starting from a volume that currently has a copy of *symmon* (verified as in Example 11-1), use *dvhtool* to extract a copy of *symmon* into a convenient spot.

```
dvhtool -v g symmon /var/tmp/symmon.IPxx
```

There is a unique version of *symmon* for each CPU module, so it is a good idea to qualify the filename with the CPU module type. Once the program is available as a normal file, you can use *dvhtool* to install it in the volume header of some other disk.

In the event there is not enough room in partition 0 (the volume header) of the target disk, it is safe to use *dvhtool* to delete the *ide* program from the volume header. The *ide* application can be booted manually from a CDROM if it is ever required.

Enabling Debugging in irix.sm

In order to make debugging symbols available in the kernel, you must make two changes, one required and one optional, in the file */var/sysgen/system/irix.sm*. As superuser, make a hard link to the file */var/sysgen/system/irix.sm* as *irix.sm.nondebug*. This enables you to return easily to a nondebugging kernel.

Including Symbols in the Kernel Image

Edit */var/sysgen/system/irix.sm*. Near the end, note the lines that resemble the following:

```
* Compilation and load flags
*   To load a kernel that can be co-resident with symmon
*   (for breakpoint debugging) replace LDOPTS
*   with the following.  You must also INCLUDE prf and idbg.
*
*LDOPTS: -non_shared -N -e start -G 8 -elf -woff 84 -woff 47 -woff 17
-mips2 -o32  -nostdlib  -T 88069000
```

The active LDOPTS statement (the one without an initial asterisk) appears a few lines later. Remove the asterisk from the front of the debugging LDOPTS to make it active. Insert an asterisk to convert the original LDOPTS into a comment.

Including idbg in the Kernel Image

The symbol-display routines used by the command-line kernel display tool, *idbg*, are contained in optional kernel modules. (See "Using idbg" on page 262.) You can change */var/sysgen/system/irix.sm* so that support for *idbg* is always present in the kernel. Alternatively, you can load these modules manually with *ml* before you use them (see the ml(1) reference page).

If you are entering an extended debugging period, make the modules permanent. Look for the lines in */var/sysgen/system/irix.sm* that resemble the following:

```
*
* Kernel debugging tools (see profiler(1M) and idbg(1M))
*
EXCLUDE: idbg
EXCLUDE: dmiidbg, grioidbg, xfsidbg, xlvidbg, cachefsidbg
USE : prf
```

Change these lines, if necessary, so that both *idbg* and *prf* are marked USE, not EXCLUDE. Verify that the file */var/sysgen/boot/idbg.o* exists. It is normally installed with the debugging kernel feature.

Parts of the *idbg* support that are unique to particular filesystems are in the other modules listed in this area of *irix.sm*. Modules such as *xlvidbg* are useful to Silicon Graphics developers but are not likely to be helpful to developers of third-party drivers. However, it does no harm to change those modules from EXCLUDE to USE also.

Including Lock Metering in the Kernel Image

In addition to the display support included by the idbg modules, you can include a module that supports lock metering. This module keeps statistics on the use of of each semaphore, basic lock, and reader/writer lock, and displays the statistics through idbg commands. To enable this facility, locate the lines in */var/sysgen/system/irix.sm* that resemble the following:

```
* Required kernel modules
...
* ksync - kernel synchronization routines (mutex_lock, sv_wait,
psema...)
*   or
*   ksync_metered  - metered kernel synchronization routines
...
*
```

```
KERNEL: kernel
INCLUDE: os, disp, mem, zero
INCLUDE: ksync
EXCLUDE: ksync_metered
```

Reverse the state of the two “ksync” lines so that ksync is excluded and ksync_metered is included.

Generating a Debugging Kernel

Run the *autoconfig* command to generate a new kernel that will reflect the changes made in *irix.sm*. The result is a new kernel file, */unix.install*, that will be renamed to */unix* and used when the system is booted. This kernel can support *idbg* but is not yet ready for standalong debugging with *symmon*.

The *setsym* command copies the symbol table from a kernel file and stores it as data within the kernel, so that *symmon* can find it. After autoconfig has created */unix.install*, apply the setsym command to it, as follows:

```
#setsym /unix.install
```

If this command returns an error message about “symbol table overflow,” it means you have neglected to activate the debugging LDOPTS statement in */var/sysgen/irix.sm*.

Tip: You can use *setsym* with the *-d* option to generate a list of all symbols in the kernel being modified. The list is very long; direct it to a file for later reference.

At this time, you may wish to create a link to the current, nondebugging kernel so you can retrieve it easily. You can also return to a nondebugging kernel by restoring the original *irix.sm* file and running *autoconfig* again.

Specifying a Separate System Console

In order to use the standalone debugger, you must have an ASCII terminal as a separate system console device. Install a terminal next to the system or workstation and connect it to the first serial port. Verify its operation as a terminal by logging in to the system. You may have to modify the file */etc/inittab* so that the line for the alternate console is active (see the inittab(4) reference page).

When you have verified that the terminal works, use the *nvram* command to change the nonvolatile RAM variable console from a letter “g” to a letter “d,” as follows:

```
# nvram console
g
# nvram console d
# nvram console
d
```

The *nvram* command is used to report and change the contents of the nonvolatile RAM variables used by the boot PROM and standalone shell (see the nvram(1) reference page).

Verifying the Debugging Tools

After performing the preceding steps, restart the system. Messages from *sash* appear on the attached terminal, rather than on the graphics screen. If *symmon* is present, it announces itself on the console terminal also.

To verify operation of *idbg*, issue the idbg command and display the process list:

```
# idbg
idbg> plist
active process list:
34:672:"xdm" pri(60) SLEEP flags: load uload siglck recalc sv
0:0:"sched" ndpri(39) SLEEP flags: sys nwake load uload sv
31:193:"inetd" pri(60) SLEEP flags: load uload siglck recalc sv
...
```

To verify operation of *symmon*, press control-A at the console terminal. The prompt string DBG: should appear. At this time the system is frozen and no longer responds to mouse or keyboard input. Type the letter c (for continue) and press return. The system returns to life.

Producing Diagnostic Displays

Normally a device or STREAMS driver produces display output in only two cases:

- To advise the operator or administrator of a serious problem.
- To display debugging information during software development.

Both of these purposes are served by the **cmn_err()** function. It brings to a kernel-level module the abilities that a user-level process gets from **printf()** and **syslog()**.

Using cmn_err

The details of **cmn_err()** usage are in the cmn_err(D3) reference page. The function prototype and the constant values it uses are declared in *sys/cmnerr.h*.

In summary, **cmn_err()** takes two or more arguments:

- A severity code that specifies how the message should be treated when it is written to the system log.
- A message string, which can have substitution points in the style of **printf()**.
- As many numeric values as are needed to substitute into the message string.

The first character of the message string specifies the destination of the message, either an in-memory buffer or the system log, or both.

Displaying to the System Log

The message is sent to the the system log daemon whenever the first message character (after substitution) is not an exclamation mark ("!"). The message is written only to the system log when the first message character is a circumflex ("^").

This is basically the same service that a user-level process receives from the **syslog()** function. (Compare the syslog(3) and cmn_err(D3) reference pages, and examine the *sys/cmnerr.h* header file; the relationship is clear.) The first argument to **cmn_err()** is a severity code which corresponds to one of the severity codes supported by **syslog()**: CE_WARN equals LOG_WARN, and so on.

Use **cmn_err()** to write log messages to record serious errors (with CE_ALERT severity) or to advise the administrator of conditions that should be changed (using CE_NOTE).

Displaying to the Circular Message Buffer

The message is stored in the next available position in a circular buffer in kernel memory whenever the first message character (after substitution) is not a circumflex ("^"). The message is stored only in the memory buffer when the first message character is an exclamation mark ("!").

The name of the circular buffer (as a symbol to *idbg* or *symmon*) is *putbuf*. The contents of *putbuf* can be displayed with the *pb* command of either *idbg* or *symmon* (see"Using symmon" on page 252 and "Using idbg" on page 262), or in a post-mortem dump using *icrash* (see "Using icrash" on page 269). Use **cmn_err()** to store debugging trace data in the circular buffer, and extract it after a stop or breakpoint with *symmon*, or use *idbg* to look at it while the system is running.

The size of the buffer can be configured by changing the kernel variable *putbufsz*, as shown in the dialog in Example 11-2.

Example 11-2 Setting Kernel putbuf Size

```
# systune -i
Updates will be made to running system and /unix.install
systune-> putbufsz
        putbufsz = 1024 (0x400)
systune-> putbufsz 4096
        putbufsz = 1024 (0x400)
        Do you really want to change putbufsz to 4096 (0x1000)? (y/n)
y
In order for the change in parameter putbufsz to become effective,
reboot the system
systune-> quit
```

Using cmn_err() Through Macros

The inventive C programmer can think of many ways to invoke **cmn_err()** using macros. One method is illustrated in the example driver displayed in Chapter 12, "Driver Example." It contains the code shown in Example 11-3.

Example 11-3 Debugging Macros Using cmn_err()

```
#ifdef DEBUG
#define DBGMSG0(s) cmn_err(CE_DEBUG,s)
#define DBGMSG1(s,x) cmn_err(CE_DEBUG,s,x)
#define DBGMSG2(s,x,y) cmn_err(CE_DEBUG,s,x,y)
#define DBGMSG3(s,x,y,z) cmn_err(CE_DEBUG,s,x,y,z)
#else
#define DBGMSG0(s)
#define DBGMSG1(s,x)
#define DBGMSG2(s,x,y)
#define DBGMSG3(s,x,y,z)
#endif
```

Calls to the DBGMSG macros evaluate to nothing unless the DEBUG variable is defined to the compiler. Macro use could be more elaborate. For example, suppose that you want every debugging macro to start with the same string, and that you do not want to have the tedium of coding the newline at the end of every debug message text. You could write macros like the one shown in Example 11-4.

Example 11-4 More Elaborate Debugging Macro

```
#ifdef DEBUG
char *_dbg_prefix = "mydriver: %s\n";
#define DBGPREFIX(s) cmn_err(CE_DEBUG,_dbg_prefix,s)
...
#else
#define DBGPREFIX(s)
...
#endif
```

Using printf()

You can call the **printf()** function from a kernel module. The kernel version of **printf()** is basically a call to **cmn_err()** with severity CE_CONT. In general it is better to use **cmn_err()** explicitly.

Using ASSERT

The **assert()** macro is familiar to many C programmers; it terminates a program with a message if its argument evaluates to false (see the assert(3X) reference page). This normal assert**()** macro does not work in a kernel module because the normal C library is not available. However, a similar function is available as the ASSERT() macro in the header file *sys/debug.h*.

The ASSERT() macro compiles to null code unless the compiler variable DEBUG is not only defined, but defined as YES. When it compiles to executable code, ASSERT() tests its argument. If the argument evaluates to false, a kernel panic is forced.

Clearly ASSERT() must be used with care, testing conditions that are truly essential to the integrity of the system. When reporting conditions that are merely operational errors, use a call to **cmn_err()** with the CE_WARN option.

Using symmon

The *symmon* program is a standalone debug monitor that can display and modify memory, and stop, start, and trace execution, without using any kernel facilities. Using *symmon* you can set breakpoints in your driver, single-step its execution, and display the contents of driver and kernel variables.

The facilities of *symmon* are unsophisticated compared to the high-level debuggers you might use to debug a user-level application. For example, *symmon* does not understand C syntax, so it cannot display data structures as structures. Execution tracing is done at the level of machine instructions, not at the level of C statements.

However, you can use *symmon* to examine the operations of a kernel module in a running system, and resume execution of the system. This is an invaluable facility when debugging a new driver.

How symmon Is Entered

When the system boots a debugging kernel with *symmon* installed, control can pass into the debug monitor under several different circumstances:

- Early in the bootstrap process, if certain environment variables are set in the stand-alone shell (see “Entering symmon at Boot Time” on page 254).
- Whenever a control-A character is typed at the system console terminal.
- Whenever a breakpoint is reached or a watchpoint is tripped (see “Commands to Control Execution Flow” on page 257).
- Whenever a kernel module calls the kernel function **debug**(*uchar_t* **msg*).
- When a non-maskable interrupt (NMI) is detected.
- When a kernel panic is detected or forced with **cmn_err()**.

When *symmon* gains control, it displays its “DBG:” prompt at the console terminal and waits for a command.

To resume execution at the point of interruption, enter the *c* (continue) command.

Using symmon in a Uniprocessor Workstation

In a single-processor workstation such as the Indy or Indigo^2, no IRIX execution takes place while *symmon* is running. The mouse and keyboard are unresponsive. (One keystroke may be stored in the keyboard hardware to be processed when the system resumes execution.) As a result, time-dependent processes can fail; for example, the system clock is not updated. Network interrupts are not taken, so if the workstation is acting as an NFS server, it will appear to be dead to other systems.

Using symmon in a Multiprocessor Workstation

In a multiprocessor, the CPU that was interrupted runs *symmon* and nothing else. For example, the CPU that executes the breakpoint, or the CPU that handles the DUART interrupt that returns the control-A character, or the CPU in which **debug()** was called, comes under the control of *symmon*. Other CPUs continue to execute normally. However, if the *symmon* CPU holds a lock, other CPUs may come to a halt waiting for the lock to be released.

The *symmon* breakpoint table is shared by all CPUs. A breakpoint set from one CPU can be taken by another CPU, or by multiple other CPUs. It is possible to run multiple instances of *symmon* concurrently. The output from all instances of *symmon* is multiplexed onto the system console terminal. However, only one CPU at a time issues the DBG: prompt. Use the *cpu* command with no argument to find out which CPU is prompting. Use the *cpu* command with a cpu number to switch to a different CPU. (See "Commands to Control Execution Flow" on page 257.)

Tip: To make multiprocessor execution more predictable, reduce the number of available CPUs. Use the PROM monitor to disable one or more CPUs before bootstrap.

Entering symmon at Boot Time

You can cause the kernel to stop during initialization and enter symmon during the bootstrap process. In order to do this, you must use the miniroot to set environment variables.

1. Restart the system, for example by giving the commands *sync* and *halt*. Eventually, the 5-item PROM menu is displayed at the console terminal.
2. Select item 5, "Enter the Command Monitor."
3. Set one or both of the environment variables *dbgstop* and *symstop* to 1, using commands such as the following:

   ```
   >> setenv symstop 1
   ```
4. Return to the PROM menu by entering the command *exit*.
5. Select menu item 1, "Start System."

In either case, *symmon* seizes the system and displays its DBG: prompt at the system console during bootstrap. When the *dbgstop* variable is set, *symmon* takes control of the system very early in the bootstrap process. Symbolic names are not initialized at this point. However, breakpoints can be set and memory can be displayed using explicit addresses.

When the *symstop* variable is set, *symmon* takes control after symbols are defined, but before driver initialization is begun. At this stop, you can display memory and set breakpoints based on entry point names of your driver.

Commands of symmon

The exact set of commands supported by *symmon* changes from release to release and from CPU model to CPU model. Many *symmon* commands are useful only to Silicon Graphics engineers who are debugging hardware and kernel problems. For a complete list of commands, see the symmon(1M) reference page, or enter *symmon* and give the *help* command. You can use control-S and control-Q on the console terminal to pause the scrolling display.

The commands described in this section are generally useful and are available on all CPU models under IRIX 6.2. These commands can be grouped into the following categories:

- Conversion between symbols and memory addresses.
- Execution control, including commands for stopping, starting, and setting breakpoints.
- Display and modification of memory, including the display of machine registers and of system data structures such as the *buf_t* and *proc_t* objects.
- Management of the virtual memory system and the TLB.

Syntax of Command Elements

The *symmon* commands all have the same form: a keyword, usually followed by one or more arguments separated by spaces.

Many commands take an address value. An address argument value can have one of the following forms:

Decimal number	A number starting with 1-9 is a decimal number, for example *4095*.
Octal number	A number starting with 0 and a digit is an octal number, for example *033*.
Hex number	A number starting with 0x is a hexadecimal number, for example *0xffff8000*.
Binary number	A number starting with 0b is a binary number, for example *0b0100*.

Symbol	A word starting with a non-digit is looked up as a symbol in the kernel symbol table, and its address is the value; for example *dk_open*.
Register	A word starting with “$” is taken as a register name, and the contents of the register as of the last interrupt is used as the argument value; for example *$a2*.
Value and offset	A value plus or minus a number is a value, for example *$a2-0x100* or *dk_open+128*.

Some commands accept a range of addresses. A range can be written in one of two ways:

- As *value1*:*value2*, meaning an inclusive range of addresses from *value1* through *value2*, for example *prtbuf:prtbuf+4096*.
- As *value1*#*count2*, meaning a range of count2 bytes beginning at value1, for example *prtbuf#4096*.

The register names that *symmon* accepts and shows in various displays are the conventional names used in MIPS assembly language programming. Refer to the *MIPSpro Assembly Language Programmer's Guide* and the processor manuals listed under “Additional Reading” on page xxxix.

Commands for Symbol Conversion and Lookup

The commands summarized in Table 11-1 are used to convert between symbolic names and their corresponding addresses.

Table 11-1 Commands for Symbol Conversion and Lookup

Command	Example	Operation
hx *name*	**hx dk_read** dk_read 0xffffffff882b0510	The name is looked up on the symbol table and if it is found, its address is displayed.
lkaddr *addr*	**lkaddr 0x882b0510** 0x882af910 lockdisptab 0x882b0510 dk_read 0x882b051c dk_write	Symbols near to the specified *addr* are listed. Use this command to find out the symbolic location of an unexpected stop.

Table 11-1 (continued) Commands for Symbol Conversion and Lookup

Command	Example	Operation
lkup *letters*	**hx dk_rea** 0x880d5f10 dk_readcap 0x882b0510 dk_read 0x332b0528 dk_readcapacity	Every symbol that contains the specified *letters* at any point is listed. There is no way to anchor the search to the beginning or end of the name.
msyms *ident*	**msyms 13** Symbols for module 13 (prefix tcl) tclinit 0xc0403d9c tclmversion 0xc0405fe0	The symbols for the loadable module *ident* are listed. Use the *ml* command with no arguments to list all modules and their ident numbers.
nm *addr*	**nm 0xc0403da0** 0xc0403da0 tclinit+0x4	The symbol nearest to the specified *addr* is listed.

Note: When symmon displays an address it normally shows a full 64 bits. In a 32-bit kernel, the most-significant 32 bits of a kernel virtual address are all-binary-1, from extension of the sign bit of the 32-bit address—as shonw in the example of hx in Table 11-1. When you enter an address to a command in a 32-bit system, you need to type only the significant 32-bit value.

Commands to Control Execution Flow

The commands summarized in Table 11-2 are used to stop, start, and single-step execution in the kernel.

Table 11-2 Commands to Control Execution

Command	Example	Operation
brk	**brk**	List all breakpoints currently set.
brk *addr*	**brk dk_read**	Set a breakpoint at the specified *addr*.
c	**c**	Restart execution at the point of interruption in the current CPU.
c *cpuid* [*cpuid*]... **c all**	**c 0**	Restart execution in the specified CPU, or in all stopped CPUs. Available in multiprocessors only.

Table 11-2 (continued) Commands to Control Execution

Command	Example	Operation
call *addr* [*args*]	**call geteminor 0**	Call a kernel function and report the contents of the result register on return.
cpu	**cpu**	Displays the cpu ID of the currently-executing CPU. Available in multiprocessors only.
cpu *cpuid*	**cpu** 0	Force *symmon* execution to the specified CPU. That CPU must be executing *symmon*. Other CPUs executing *symmon* wait. Available in multiprocessors only.
goto *addr*	**goto geteminor**	Set a temporary breakpoint at *addr* and then continue execution as for the *c* command (in effect "go until *addr* is reached").
quit	**quit**	Return to the boot PROM, forcing an instant reboot.
s [*count*]	**s 8**	Single-step through 1 or *count* instructions, displaying each instruction and the register contents it uses. A branch and the instruction in "delay slot" following it count as 1. Steps into subroutines.
S [*count*]	**S 8**	Single-step through 1 or *count* instructions as for the *s* command, but do not step into subroutines.
unbrk *n*	**unbrk 2**	Remove break point number *n*. Use *brk* with no argument to list break points by number.
wpt {r | w | rw} *physaddr*	**wpt r 0x0841f608**	Set a hardware watchpoint on a physical address.

Tip: One way to force a memory dump from *symmon* is the command *call dumpsys*.

Following a break or a watchpoint, use the *bt* command to display the stack history and use *printreg* to display the registers (see "Commands to Display Memory" on page 260).

The hardware watchpoint used by the *wpt* command uses hardware registers in the MIPS R4000 and R10000 processors (the R8000 does not support the watchpoint registers). When a read or write access is addressed to any byte in the doubleword specified by the physical address, *symmon* gains control and displays the instruction that is attempting the access on the console terminal.

The argument of *wpt* must be a physical memory address and a multiple of 8. Use *tlbvtop* to get the physical equivalent of an address in a user address space (see “Commands to Manage Virtual Memory” on page 259). In a 32-bit kernel, the physical equivalent of an address in kernel space is obtained by changing the most significant hex digit to 0.

Commands to Manage Virtual Memory

The commands summarized in Table 11-3 are used to display and manage the virtual memory translation system.

Table 11-3 Commands to Manage Virtual Memory

Command	Example	Operation
cacheflush *range*	**cacheflush $6:$6+4096**	Flush both the instruction and data caches when they contain data that falls in *range*.
tlbdump [*lo:hi*]	**tlbdump 1:3**	Display the contents of the TLB registers. When a range of numbers is given, the registers from *lo* through *hi*-1 are displayed.
tlbflush [*lo:hi*]	**tlbflush**	Flush (nullify) the TLB registers specified. The registers are reloaded as required during subsequent execution.
tlbpid	**tlbpid** Current dbgmon pid = 79	Display the process slot number of the process whose context is in the TLB.
tlbvtop *addr*	**tlbptov 0xffffc000**	Display the TLB register that maps *addr*.

Commands to Display Memory

The commands summarized in Table 11-4 are used to display memory or variables.

Table 11-4 Commands to Display Memory

Command	Example	Operation
bt [*frames*]	**bt 4**	Display the calling function, the arguments, and the name of the called function for up to *frames* stack frames. Most useful after a break or interrupt.
dis *range*	**dis geteminor**	Disassemble and display the instructions over the specified range.
dump [-b|-h|-w] [-o|-d|-x|-c] *range*	dump 0xc0000000	Display memory over a specified range. The options -b, -h, and -w specify how memory is grouped, as units of 1, 2, or 4 bytes. The options -o, -d, -x, and -c specify translation into octal, decimal, hex and character.
kp [*routine*]	**kp plist**	Invoke a kernel print routine loaded with the idbg kernel module. If no routine is given, all available names are displayed.
printregs	**printregs**	Display all the registers as they were when the debugger was entered.
string *range* [*max*]	**string $v1 0x80**	Display memory as an ASCII string in quotes. Display stops at the first null byte, or, when *max* is specified, after at most *max* bytes.

The display routines available to the *kp* command are discussed under "Using idbg" on page 262. The names that *idbg* accepts as commands are all available under *symmon* through the *kp* command.

Use the *dump* command under *symmon*. Under *idbg*, use the *hd* command for the same purpose.

Utility Commands

The commands summarized in Table 11-5 are general-purpose utilities.

Table 11-5 Utility Commands

Command	Example	Operation
calc	**calc**	Starts a simple stack-oriented calculator (see text).
clear	**clear**	Clear the screen of the system console terminal.
help	**help**	List one-line summaries of all available commands. Use control-S and control-Q to control the scrolling of the display.
g [-b|-h|-w | -d] [*addr* | $*regname*]	**g $a1** 0x882fadf8: 4294967295 0xffffffff	Display one byte, halfword, word or doubleword (default word) of memory, or the contents of one register at the time symmon was entered, in decimal and hex.
p [-b|-h|-w | -d] [*addr* | $*regname*] *value*	**p -w 0xc0000000 4095**	Write a byte, halfword, word, or doubleword (default word) into a saved register or into memory at the specified address.

Using idbg

The *idbg* command is a utility that provides much of the display capability of *symmon* but from a command line, without stopping the system. Many details of *idbg* use are covered in the idbg(1M) reference page. Keep in mind that all idbg commands are available under the standalone debugger through the kp command (see "Commands to Display Memory" on page 260).

Loading and Invoking idbg

Superuser privilege is required to invoke *idbg*, because it maps kernel memory. The command is ineffective unless its support modules have been made part of the kernel. This can be done permanently by changing the *irix.sm* file (see "Including idbg in the Kernel Image" on page 246). Alternatively, you can load the needed modules dynamically using the *ml* command, as follows:

```
# ml ld -i /var/sysgen/boot/idbg.o
```

Dynamic loading is discussed at more length in the idbg(1M) and ml(1M) reference pages.

When the support modules are loaded, idbg can be invoked in three styles.

Invoking idbg for Interactive Use

Invoking the command with no arguments causes it to enter interactive mode, prompting for one command after another from standard input, as shown in Example 11-5.

Example 11-5 Invoking idbg Interactively

```
# idbg
idbg> plist 187
pid 187 is in proc slot 31
idbg> quit
#
```

The command terminates when the command quit is entered, or when control-D (standard input end of file) is typed.

Invoking idbg with a Log File

Invoking the command with the *-r* option and a filename causes it to write all its output to the specified file, as shown in Example 11-6.

Example 11-6 Invoking idbg with a Log File

```
# idbg -r /var/tmp/idbg.save
idbg> plist 187
pid 187 is in proc slot 31
idbg> proc 31
proc: slot 31 addr 0x8832db30 pid 187 ppid 1 uid 0 abi IRIX5
 SLEEP flags: load uload siglck recalc sv
...
idbg> ^D
# cat /var/tmp/idbg.save
pid 187 is in proc slot 31
proc: slot 31 addr 0x8832db30 pid 187 ppid 1 uid 0 abi IRIX5
 SLEEP flags: load uload siglck recalc sv
...
#
```

You can use this method to collect a series of displays in a single file as you test a driver.

Invoking idbg for a Single Command

You can invoke *idbg* with a command on the command line. The output of the single command is written to standard output, where it can be captured or piped to another program. Example 11-7 shows one simple use of this feature.

Example 11-7 Invoking idbg for a Single Command

```
# idbg plist | fgrep -c tcsh
3
#
```

Since the displays of *idbg* are very rich, there are endless opportunities to use this mode to generate data within shell scripts, and to process it using tools such as *awk* and *perl*. Using *perl* you could write an intelligent display routine that showed the status of your driver's private data structures using your own terminology and display format.

Commands of idbg

Almost all *idbg* commands are concerned with displaying kernel memory data in different ways. There are commands to display almost every type of kernel data.

The vocabulary of commands changes from release to release, and can change within releases by software patches. Also, the commands available depend on which support modules are loaded; for example lock and semaphore meters cannot be displayed unless the ksynch_meter module is loaded (see "Including Lock Metering in the Kernel Image" on page 246). Only a few commands are listed in the idbg(1M) reference page.

The commands summarized in this book are generally useful and available on all platforms in the current release of IRIX. For a complete (but cursory) list, use the command itself.

```
# idbg help | lp
```

In general, commands take zero or one argument. Typically the argument is a number, which can be any of the following:

- A kernel symbol, optionally +offset
- A number in hexadecimal (starting with 0x)
- A number in octal (starting with 0)
- A number in decimal.

The number is interpreted in the context of the command: sometimes it represents a process ID (pid), sometimes a process "slot" number or a buffer number. Often commands treat positive numbers as slot numbers or table indexes, while negative numbers are treated as addresses in kernel space.

Commands to Display Memory and Symbols

The commands summarized in Table 11-6 are used to display memory based on specific addresses or symbols, and to display the addresses for kernel symbols.

Table 11-6 Commands to Display Memory and Symbols

Command	Operation
dsym *addr* [*length*]	Dump memory by words, starting at *addr*. When a word of memory data is reasonably close to the value of a kernel symbol, the symbol plus offset is displayed instead of the hex value.
hd *addr* [*length*]	Dump memory in bytes, with ASCII translation, starting at *addr*. When *length* is given, it is a count of words (not bytes) to be displayed.
pb	Display the strings in the circular putbuf (see “Displaying to the Circular Message Buffer” on page 250).
string *addr* [*max*]	Display memory as an ASCII string. Display stops at the first null byte, or, when *max* is specified, after at most *max* bytes.

When you display the circular buffer, there is no special indication to show which line is the newest. You have to deduce the boundary between the newest and oldest lines from the content.

Commands to Display Process Information

The commands summarized in Table 11-7 are concerned with displaying the status of processes. Processes are recorded in an array of “slots.” The *plist* command gives the process slot number for a given process ID. The other commands take slot numbers.

Table 11-7 Commands to Display Process Information

Command	Operation
eframe [*addr* | *slot*]	Diplay the contents of an exception frame. With no argument, displays the last exception taken for the current process. Else displays the exception associated with the process specified either by address (negative number) or process table slot number (positive number)
pchain *slot*	Display the slot numbers of sibling processes to the process in *slot*.

Table 11-7 (continued) Commands to Display Process Information

Command	Operation
plist [0 | *pid*]	With no argument, displays a one-line summary of every active process slot, including slot number and process ID. When the argument is 0, displays all inactive process slots. With a nonzero PID, displays the slot containing that process.
ptree [*addr* | *pid*]	With a *pid* (number greater than zero), finds the process structure for that process. Else uses the process structure at *addr*. Displays the command name and command arguments for that process and for all processes that descend from it.
proc [*addr* | *slot*]	Display all the fields of a process structure specified either by address (negative number) or process table slot number (positive number).
signal [*addr* | *slot*]	Display information about pending signals for the process specified either by address (negative number) or process table slot number (positive number)
slpproc [-2 | -4 | -8]	Displays a summary of all processes with p_stat of SSLEEP or SXBRK. When an argument is given, its absolute value is used as a mask: 2 ignores processes in **wait()**; 4 ignores processes without upages; 8 ignores processes on a sleep semaphore.
ubt *slot*	Display a backtrace of the call stack of the sleeping process in the specified slot.
user [*addr* | *slot*]	Display the user area associated with the process specified either by address (negative number) or process table slot number (positive number)

Commands to Display Locks and Semaphores

The commands summarized in Table 11-8 display the state of semaphores and locks of different kinds, including metering information when the metered-lock module is included in the kernel.

Table 11-8 Commands to Display Locks and Semaphores

Command	Operation
;ock *addr*	Display the state of the spinlock at *addr*. This command is available only in multiprocessor systems.
mrlock *addr*	Display the state of the reader/writer lock at *addr*.
mutex *addr*	Display the state of the mutual exclusion lock at *addr*.
sema *addr*	Display the state of the semaphore at *addr*.
smeter *addr*	Display metering information about the semaphore at *addr*. When *addr* is positive, it is taken as an index to the semaphore metering array.
sv *addr*	Display the state of the synchronizing variable at *addr*, including waiting processes and metering information.

Commands to Display I/O Status

The commands summarized in Table 11-9 can be used to display the status of an I/O device or driver.

Table 11-9 Commands to Display I/O Status

Command	Operation
file [*addr*]	When *addr* is omitted, displays a summary of all entries of the kernel table of open files. When *addr* is the address of a file structure, displays only that entry.
scsi *addr*	Display the contents of the *scsi_request* structure at *addr*.
uio *addr*	Display the contents of the *uio_t* object at *addr*.

Commands to Display buf_t Objects

The commands summarized in Table 11-10 are used to display the state of *buf_t* objects and the queue of *buf_t* objects maintained by the kernel.

Table 11-10 Commands to Display buf_t Objects

Command	Operation
buf [*addr*]	If *addr* is omitted, print the entire buffer chain. When *addr* is supplied as the address of a buf_t, dump that structure.
findbuf *blkno*	Display any *buf_t* in the buffer chain with *b_blkno* containing *blkno*.
qbuf *eminor*	Find and display all *buf_t* objects that are queued to the device with external minor number *eminor*.

Commands to Display STREAMS Structures

The commands summarized in Table 11-11 are concerned with displaying STREAMS data structures such as message buffers.

Table 11-11 Commands to Display STREAMS Structures

Command	Operation
datab *addr*	Display the contents of the STREAMS data block at *addr*.
mbuf *addr*	Display the contents of the STREAMS *mbuf* structure at *addr*.
modinfo *addr*	Display the contents of the module info structure at *addr*.
msgb *addr*	Display the contents of the STREAMS message block at *addr*.
qband *addr*	Display the contents of the *qband_t* object at *addr*.
qinfo *addr*	Display the contents of the *qinit* structure at *addr*.
strh *addr*	Display the contents of the *stdata* structure at *addr*.
strfq *addr*	Display the contents of the *queue_t* object at *addr*.

Commands to Display Network-Related Structures

The commands summarized in Table 11-12 display data structures that are related in one way or another to networking and network device drivers.

Table 11-12 Commands to Display Network-Related Structures

Command	Operation
ifnet *addr*	Display the contents of the ifnet object at *addr*.
rawcb *addr*	Display the contents of the *rawcb* structure at *addr*.
rawif *addr*	Display the contents of the *rawif* structure at *addr*.
sock *addr*	Display the *sockbuf* structure at *addr*. When *addr* is positive, it is taken as a physical address; otherwise it is a kernel address.

Using icrash

The *icrash* program is a post-mortem analysis tool for system crashes. You can use icrash to generate a wide variety of reports and displays based on a kernel panic dump from a crashed system. Study the icrash(1M) reference page for the current release. For example, you can display the *putbuf* message buffer using the *stat* command of *icrash*.