Chapter 10

Driver Installation and Testing

This chapter explains how to use **symmon**, the kernel debugger. It contains the following sections:

Note: This program requires an ASCII terminal on the first serial port.

Using the Kernel Debugger

Like all software, the IRIX kernel needs a reliable debugging tool. This chapter describes the kernel debugger, **symmon**(1), an indispensable tool for debugging device driver software. **symmon**(1) and the other kernel debugging tools are not automatically loaded during installation. To put these tools onto your system, install the subsystem *eoe2.sw.kdebug* manually from your software release CD-ROMs.

You can load **symmon**(1) coresident with a standalone program or operating system, then use **symmon**(1) to control the execution of that program or operating system. **symmon**(1) gives you commands to examine and alter memory and registers, set breakpoints, and execute programs.

Making a Debuggable Kernel

To make a kernel that you can debug from **symmon**(1), you need to create a kernel slightly different from the standard one. Use the following steps (many of which you used when you added a device driver to the kernel) to create a debuggable kernel.

1. Become root:

   ```
   % su
   ```

2. Edit */var/sysgen/system/irix.sm*.

   - Change the line:

     ```
     EXCLUDE idbg
     ```

     to

     ```
     INCLUDE idbg
     ```

   - Find the two lines containing *LDOPTS* toward the end of the file. Comment out the uncommented lines and remove the comment marker for the currently commented out lines. (To comment out a line, put an asterisk character in the first column.)
   - Make a note of the change. When you are finished debugging, undo the change so that you can make a standard kernel.

3. Copy the current kernel to a safe place before rebooting.

   ```
   # cp /unix /unix.orig
   ```
   or
   ```
   # ln /unix /unix.orig
   ```

4. Run */etc/autoconfig* and answer yes if prompted.

5. Use **halt** to bring the system down. When you issue the **halt** command, the system overwrites the current kernel, */unix*, with the kernel you have just created, */unix.install*.

   ```
   # halt
   ```

6. On IRIS-4D/100, 4D/200, 4D/300, and 4D/400 Series workstations, set the front panel dip switches as shown in Figure 10-1.

   Caution: For these workstations, you must not use any odd-numbered port connected to a CPU for data transfer because an inadvertent `<ctrl-A>` in the data stream can put the system into **symmon**. This is true of any odd-numbered port not connected to a 6-port serial card.

   Personal IRIS, Indigo, Indigo^2, Indy, Crimson, CHALLENGE/Onyx, and POWER CHALLENGE/POWER Onyx systems do not require physical switches for kernel debugging.

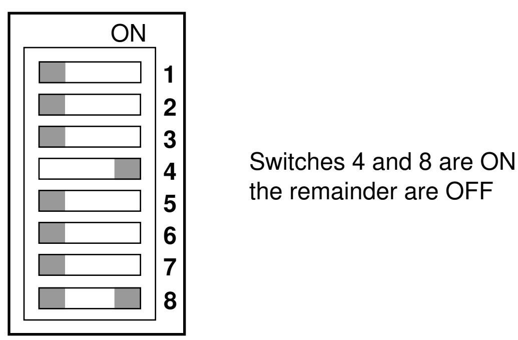

Figure 10-1 IRIS Front Panel Dip Switches

7. After halting the system, push the `<Reset>` button to force the processor to read the switches.

Making an ASCII Terminal the Console

To use the kernel debugger, you must install an ASCII terminal as the system console. If you do not install an ASCII terminal as the system console, you can still use the newly built kernel under your normal configuration, but you cannot use the kernel debugger. Some systems (IRIS-4D/20, -4D/25, -4D/30, -4D/35, Indigo, Indigo^2, Indy, and Crimson) allow you to use the graphics text port when the window system is not running. **symmon** may fail occasionally in this configuration, but it can be useful when no ASCII terminal is available.

To install an ASCII terminal as the system console:

1. Connect the terminal to Serial Port 1 for a single processor (or to the appropriate port number for multiprocessor systems).
2. Shut the system down in an orderly fashion.
3. Select the Command Monitor mode from the Bootstrap menu.
4. Type the following from the Command Monitor mode:

```
setenv console d
init
```

At this point, all console input and output occur on the ASCII terminal. Boot the system as usual. The kernel loads and, because the kernel includes the module **idbg**(), **symmon**() loads automatically from the *root* disk volume header partition for those systems where it is not part of the PROM.

Note: Although kernel output normally appears in a graphics window once graphics are started, the debugger still uses the ASCII terminal. The interactive version of **symmon**, called **idbg**, uses the same commands as **symmon** but does not stop the system from processing and does not require an ASCII terminal. You can use **idbg** to look at variables but not to set them. (See “Using symmon’s Kernel Print Command” on page 296.)

Invoking symmon

Once **symmon**(1) is loaded, control is transferred to the kernel. Control transfers to **symmon**(1) in any of the following ways:

- From the console, press **`<ctrl-A>`**

  Note: **symmon** can also be invoked from any odd-numbered CPU port on IP5 and IP7 systems, except for the IRIS-4D/210.

- Set the *dbgstop* environment variable from command mode before booting the kernel. (You cannot use kernel symbols at this point, but all other functions work.) To do so, set the *dbgstop* environment variable:

  ```
  > setenv dbgstop 1
  ```

  This transfers control to **symmon**(1) early in system startup. (Symbolic debugging is not enabled at this point.)

- Set the *symstop* environment variable from command mode before booting the kernel:

  ```
  > setenv symstop 1
  ```

  This transfers control to symmon(1) after the symbol table has been loaded but early in the system startup procedure.

- Insert a call to the **debug(uchar_t *msg**) service from a known location within your kernel.
- The IRIX kernel executes a breakpoint instruction.
- A system panic occurs.

Displaying and Changing Registers

symmon(1) provides commands that allow you to display and alter the processor and coprocessor general-purpose registers. To identify a general-purpose register (there are 32 registers, numbered 0 through 31), you can use names such as “r0” or “r31,” or you can use the compiler usage names (in some cases, you may need to prepend a “$”). The compiler names and the associated “r” names are listed in Table 10-1.

Table 10-1 Processor and Coprocessor General-purpose Registers

Compiler	Processor	Usage
zero	r0	Wired zero
at	r1	Assembler temporary
v0	r2	Function value registers
v1	r3	
a0	r4	Argument registers
a1	r5	
a2	r6	
a3	r7	
t0	r8	Caller saved registers
t1	r9	
t2	r10	
t3	r11	
t4	r12	
t5	r13	
t6	r14	
t7	r15	
s0	r16	Callee saved
s1	r17	

Table 10-1 (continued) Processor and Coprocessor General-purpose Registers

Compiler	Processor	Usage
s2	r18	
s3	r19	
s4	r20	
s5	r21	
s6	r22	
s7	r23	
t8	r24	Caller saved
t9	r25	
k0	r26	Kernel temporary
k1	r27	
gp	r28	Global pointer
sp	r29	Stack pointer
fp/s8	r30	Callee saved
ra	r31	Return address

You can refer to special R2000/3000/4000/8000 registers and system coprocessor registers by using the names listed in Table 10-2, Table 10-3, and Table 10-4.

Table 10-2 R2000-R4000 Processor and Coprocessor Special Registers

Name	R2000/3000/4000 Register
mdlo	Mul/div register lower word
mdhi	Mul/div register higher word
pc epc	Exception PC
sr	Status register
cause	Cause register
tlbhi entryhi	TLB entry hi register
tlblo entrylo	TLB entry lo register
badvaddr	Bad virtual address
index inx	TLB index register
context ctxt	Context register
random	Random register

Table 10-3 R4000-only System Coprocessor Registers

Name	R4000Series System Coprocessor Register
tlblo0	TLB entrylo0 register
tlblo1	TLB entrylo1 register
pagemask	TLB pagemask register
wired	TLB wired register
count	Timer count register
compare	Timer compare register
watchlo	WatchLo register
watchhi	WatchHi register

Table 10-3 R4000-only System Coprocessor Registers

Name	R4000Series System Coprocessor Register
ecc	Ecc register
cacherr	Cache error and status register
errepc	Cache ErrorEpc register
taglo	Cache tag register
config	Configuration register

Table 10-4 R8000-only Special Registers

Name	R8000 Series Special Register
tlbset	TLBset register (index into a TLB entry's set)
trapbase	Trapbase register (base address of trap vectors)
ubase	UBase register
pbase	PBase register
gbase	GBase register
shiftamt	ShiftAmt register
wired	Wired register
badpaddr	BadPAddr register

Using symmon Commands

This section describes the **symmon** commands. "symmon's dbgmon Mode Commands" describes the general **symmon** commands, which are referred to as **dbgmon** mode commands. "Using symmon's Kernel Print Command" describes the **symmon** command **kp**. The **kp** command briefly accesses another mode of the kernel debugger, *Kernel Print* mode (also called *KP* mode). After **symmon** executes the **kp** command, it returns to **dbgmon** mode.

Use **symmon**'s *dgbmon* mode when you need ordinary debugger commands such as **brk**, **dump**, **get**. Use **symmon**'s *KP* mode when you want to view kernel structures and other information.

Help for symmon Commands

To see a listing of messages concerning the **symmon** commands while running **symmon**, type a question mark (**?**) and press **<Enter>**. In addition, commands that require arguments give a usage summary if you enter the command without arguments.

symmon's dbgmon Mode Commands

This section describes the commands of **symmon**'s *dbgmon* mode. Each heading lists the name of the command, its option flags, and its arguments. The headings use square brackets to indicate optional arguments and option flags. Following each heading is text that describes the purpose of the command. Following the command description are descriptions of each argument or option flag (if any). Different systems and operating system releases may have slightly different commands or options. For example, **symmon** is part of the PROM on some systems, while it is loaded at boot time on others. (See Table 10-5.)

Table 10-5 symmon's dbgmon Mode Commands

Command	Description
brk [*addresslist*]	Mark breakpoints at specified addresses.
bt	Show the backtrace leading up to entry to the debugger. Also see the **kp ubt** command.
c	Leave the debugger environment and continue execution.
cacheflush *range*	Flush both the instruction and the data caches over the range of addresses given.
clear	Clear the screen.
call [*pc*] [*arg1, arg2, arg3, arg4*]	Execute the code starting at the address specified.
dis *range*	Disassemble MIPS assembly instructions for the specified range of memory locations.

Table 10-5 symmon's dbgmon Mode Commands

Command	Description
dump [-*Bcdoux*] [-*bhw*] *range*	Get a formatted display of an area of memory.
g [-*bhw*] *location*	Display the contents of a memory location or a register.
goto *list*	Continue execution of the client process from the location indicated by the client pc register to the location specified.
help	List a short summary of the built-in commands.
hx *namelist*	Convert a name or list of names into their equivalent hexadecimal address values.
lkaddr *address*	Print symbols "near" the given address.
lkup *name*	Print address of the specific partial name.
nm *addresslist*	Display the equivalent symbol name of a hexadecimal address or list of hexadecimal addresses.
p [-*bhw*] *location value*	Set the contents of the register or memory location to a value.
s [*count*] *and S* [*count*]	Execute one or more instructions of client code.
sleep	Put a processor into the waiting loop on multiprocessing systems.
string *address* [*maxlen*]	Display memory as a null-terminated ASCII string.
tlbdump [*range*]	Display the current contents of an R2000/3000/4000 address translation buffer.
tlbflush [*range*]	Flush mappings from R2000/3000/4000 address translation buffer.
tlbmap [-*i index*] [-*ndgv*] *vaddress paddress*	Establish a virtual-to-physical address map in the R2000/3000/4000 translation buffer.
tlbpid [*pid*]	Get or set the process identifier (*pid*).
tlbptov *physaddr*	Display the *tlb* entries that map a physical address.
tlbvtop *vaddress* [*pid*]	Display the R2000/3000/4000 translation buffer entries that map the specified virtual address.

Table 10-5 symmon's dbgmon Mode Commands

Command	Description
unbrk *bpnumlist*	Remove breakpoints.
wake	Wake up slave processors.
wpt [*r*|*w*|*rw*] [*0*|*physaddr*]	Set a read, write, or read/write watch point at physical address *physaddr*, using the R4000 watch point registers.

brk Command

Use **brk** to mark breakpoints at specified addresses. If you do not specify an argument, **brk** shows all currently set breakpoints.

Synopsis

`brk [`*addresslist*`]`

Arguments

addresslist Use this parameter to specify the addresses at which you want breakpoints. You can enter addresses either numerically or symbolically.

If you enter the addresses numerically, **brk** assumes that all address values are in base 10 unless you specify otherwise. To enter a hexadecimal value, precede the value with **`0x`**. To enter an octal value, precede the value with **`0`**.

If you enter the addresses as symbolic names, enter the addresses according to the format *symbol* or *symbol+hexval*, where *hexval* is a word-aligned hexadecimal value.

bt Command

bt shows the backtrace leading up to the entry to the debugger. However, you will probably find the information from *kp ubt* to be more useful. In general, backtraces do not go past the last interrupt or exception.

Synopsis

```
bt
```

c Command

c allows you to leave the debugger environment and continue execution. *c* directs **symmon** to continue execution from the location indicated by the current value of the client program counter register. This command is the counterpart of the **`<ctrl-a>`** character combination that returns control to the debugger. See also *q*(uit).

Synopsis

```
c
```

call Command

call executes the code starting at the address specified.

Synopsis

`call [pc] [`*arg1 arg2 arg3 arg4*`]`

Arguments

[*pc*]	Use this argument to specify the starting address of a client routine. If no argument is specified, the saved *pc* is used.
[*args*]	Use these optional arguments to specify the arguments (up to four) that you want to pass to the routine pointed to by pc.

dis Command

dis disassembles MIPS assembly instructions for the specified range of memory locations.

Synopsis

`dis` *range*

Arguments

range Use this argument to specify the range of memory you want to display. You can specify the *range* argument in one of the following three formats:

baserange is a base address. Use this format to disassemble the contents of a single location. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset.

base#count range is a base address, followed by a number "#" character, followed by a count value. Use this format to disassemble a range of *count* words, starting at the base address. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. However, the *count* value is a hexadecimal value.

base:limit range is a base address, followed by a colon ":" character, followed by an upper limit address. *base* is a hexadecimal address or a symbol name. Use this format to disassemble the contents of those words whose addresses are greater than or equal to the *base* address, but less than the *limit* address. The value given as the *base* address or as the *limit* can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset.

Note: In all the formats described above, the *base* address must be word-aligned.

dump Command

Use **dump** to get a formatted display of an area of memory.

Synopsis

`dump [–`*Bcdoux*`] [–`*bhw*`]` *range*

Arguments

[-*Bcdoux*] Use these options to set the format in which dump displays the contents of a memory location. The default format is hexadecimal. The formats associated with these options are:

x = hexadecimal
o = octal
d = decimal
u = unsigned decimal
c = ASCII
B = binary

[-*bhw*] Use these options to specify the size of the memory location. The default is word. The associated sizes are:

b = byte
h = half-word
w = word

range Use *range* to specify the amount of memory to be displayed. You can specify range in one of the following formats:

base range is a base address. Use this format to disassemble the contents of a single location. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset.

base#count range is a base address, followed by a number "#" character, followed by a count value. Use this format to disassemble a range of *count* words, starting at the base address. The base address can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset. However, the *count* value is a hexadecimal value.

base:limit range is a base address, followed by a colon ":" character, followed by an upper limit address. *base* is a hexadecimal address or a symbol name. Use this format to disassemble the contents of those words whose addresses are greater than or equal to the *base* address, but less than the *limit* address. The value given as the *base* address or as the *limit* can be a hexadecimal address, a symbol name, or a symbol name plus a hexadecimal offset.

g Command

Use **g** (or **get**) to display the contents of a memory location, general-purpose register, special-purpose register, or a system coprocessor register.

Synopsis

`g [-`*bhw*`]` *location*

Arguments

[-*bhw*] Use these options to specify the size of the location you want to display. The default size is a word. The sizes associated with these options are:

b = byte
h = half-word
w = word

location Use *location* to specify the location or register you want to get. The format for a location or register can be a hexadecimal value or the symbolic name (plus a hexadecimal offset) of the address or the client register you want to display.

To specify one of the 32 general-purpose registers, 0 through 31, use the names **r0** through **r31**, or use the compiler-usage names. See Table 10-1 for a list of such registers.

goto Command

goto continues the execution of the client process from the location indicated by the client program counter (pc register) to the instruction at the location(s) you specify. Use this command to set a list of temporary breakpoints.

Synopsis

`goto` *list*

Arguments

list	Use this parameter to specify a list of hexadecimal addresses and/or names of locations at which you want to set temporary breakpoints. These temporary breakpoints are automatically removed once they have been encountered.

hx Command

Use **hx** to convert a name or list of names into their equivalent hexadecimal address values.

Synopsis

`hx` *namelist*

Arguments

namelist	Use this argument to specify the list of names you want to convert. This argument can be one or more symbol names (plus hexadecimal offset).

The next two items (**lkaddr** and **lkup**) are not available in older versions:

lkaddr Command

lkaddr prints symbols "near" the given address.

Synopsis

`lkaddr` *address*

Arguments

address	Use this argument to specify the starting address of the string you want to display.

lkup Command

name may be only a partial name, such as "init.". Symbols containing *name* are printed with their respective addresses.

Synopsis

`lkup` *name*

Arguments

name	*name* is a filename.

nm Command

Use **nm** to display the equivalent symbol name (plus hexadecimal offset) of a hexadecimal address or list of hexadecimal addresses.

Synopsis

`nm` *addresslist*

Arguments

addresslist	Use this parameter to specify the addresses at which you want breakpoints. You can enter addresses either numerically or symbolically.

p Command

Use *p (*or *put*) to set the contents of the register or memory location to a value.

Synopsis

`p [-bhw]` *location value*

Arguments

[-*bhw*]	Use these options to specify the size of the memory location or register value you want to set. The default size is word. The sizes associated with each of these options are: *b* = byte *h* = half-word *w* = word
location	Use this argument to specify the memory location or register you want to set. To specify a memory location, use the hexadecimal name of the memory location, or the symbolic name (plus a hexadecimal offset) of a memory location you want to set. To specify any of the 32 general-purpose registers, use the names *r0*, *r1, r2* through *r31*, or use the compiler mnemonics for these registers (see Table 10-1). You can also set special-purpose registers and the system coprocessor registers (see Table 10-2). To specify the pc register, use the entry point of a client routine as the *location a*rgument.
value	Use this argument to specify the hexadecimal value you want to write to the specified memory location or register (*value* is always assumed to be hexadecimal.)

sleep Command

Use **sleep** to put a processor into the waiting loop on multiprocessing systems. To awaken the sleeping processor, use the **wake** command. On IRIS-4D 100/200/300/400 Series workstations, do not use this command on CPU 0.

Synopsis

```
sleep
```

s [count] and S [count]

Both **s** and **S** allow you to execute one or more instructions of client code. These two commands differ only in the way they handle *jal* and *bal*, instructions that execute subroutines.

When **S** executes an instruction that calls a subroutine, it executes the entire subroutine as a single instruction, up to and including the *return* instruction. **S** fails to regain control if the subprocedure does not return.

When **s** executes an instruction that calls a subroutine, it counts the jump to the subroutine as a single instruction, then executes instructions within that subroutine up to the number you have specified (minus the one counted when executing the jump instruction). If you specify only one instruction (the default), and the next instruction calls a subroutine, **s** jumps to the subroutine and stops.

Synopsis

```
s [count]
S [count]
```

Arguments

[*count*]	Use this optional argument to specify the number of instructions you want s or S to execute. The default is 1.

string Command

Use this command to display memory as a null-terminated ASCII string. This argument escapes non-printable characters with the backslash character, just as is done in the C programming language.

Synopsis

`string` *address* `[`*maxlen*`]`

Arguments

address	Use this argument to specify the starting address of the string you want to display.
[*maxlen*]	Use this optional argument to specify the length of the string. The default value for this argument is 70.

tlbdump Command

Use **tlbdump** to display the current contents of an R2000/3000/4000 address translation buffer.

Synopsis

`tlbdump [`*range*`]`

Arguments

[*range*] Use this optional argument to select a range of tlb entries. The default behavior is to display the entire contents of tlb. The three formats for range are:

baserange is a hexadecimal value. An index into *tlb. tlbdump* displays the contents of this single location.

base#countrange is a table index followed by a number "#" character, followed by a count value. **tlbdump** displays the contents of a range of *count tlb* entries starting at the *tlb* entry whose index is *base*. Both *base* and *count* are hexadecimal values.

base:limitrange is a table index followed by a colon ":" character, followed by an upper limit index. Use this format to display a range of *tlb* entries whose indices are greater than or equal to *base*, but less than *limit*. These index values are all hexadecimal values.

tlbflush Command

Use **tlbflush** to flush mappings from the R2000/3000/4000 address translation buffer, thus making them invalid, not matching any possible **address/pid** pair. You can use this command to clear translations for both **symmon** and the client process.

Synopsis

`tlbflush [`*range*`]`

Arguments

[*range*] Use this optional argument to specify the range of tlb entries that you want to flush. If you do not specify a range, tlbflush defaults to clearing all tlb entries. To specify a [*range*], use one of the following formats:

Note: Multiprocessor systems have an address translation buffer associated with each processor. A command that dumps or changes translation buffers affects only the processor associated with the debug terminal from which such commands issued.

tlbmap Command

Use **tlbmap** to establish a virtual to physical address map in the R2000/3000/4000 translation buffer. You can use **tlbmap** to establish mappings for both **symmon** and the client process.

Synopsis

`tlbmap [-i` *index*`] [-`*ndgv*`] [-`*dgv*`] [-c` *algo*`]` *vaddress paddress*

Arguments

[-*i index*] Use the -i option to specify the particular tlb entry, index, that you want to use to contain the mapping. If you do not specify a *tlb* index, **tlbmap** uses a random *tlb* entry, at an index ranging 8 to 63.

[-*ndgv*] (R2000/3000 only)

[-*dgv*] (R4000 only)

Use these options to set bits in the *tlb* entry for the map. Each option controls a single bit. By default, these bits are unset (zero). The significance of setting a bit (and the associated options) are:

n = non-cacheable (R2000/3000 only)
d = data
g = global
v = valid

[-*c algo*] (R4000 only)

Use this to set the cache algorithm in the *tlb* entry for the map. The algorithms are specified by a number. The designations are:

0: reserved
1: reserved
2: uncached
3: cacheable, non-coherent
4: cacheable, coherent exclusive
5: cacheable, coherent exclusive on write
6: cacheable, coherent update on write
7: reserved

vaddress — Use this argument to specify the virtual address side of the map.

paddress — Use this argument to specify the physical address side of the map.

tlbpid Command

Use **tlbpid** to get or set the process identifier (*pid*), a value in the R2000/3000/4000 system coprocessor register, **tlbhi**. The **tlbpid** command affects only the process identifier used by **symmon** and client code executed through the **call** command. **tlbpid** does not affect the process identifier used when client code is executed by single stepping or when continuing execution.

Synopsis

`tlbpid [`*pid*`]`

Arguments

[*pid*] — Use this optional argument to indicate the value to which you want **tlbpid** to set the *pid* value in the R2000/3000/4000 coprocessor register, **tlbhi**.

If you specify no [*pid*] argument, **tlbpid** displays the *pid* value currently in the R2000/3000/4000 coprocessor register, **tlbhi**.

tlbptov Command

Use **tlbptov** to display the *tlb* entries that map a physical address. **tlbptov** searches the R2000/3000/4000 translation buffer, looking for translations that map the specified physical address. **tlbptov** displays all matches, whether valid or invalid.

Synopsis

`tlbptov` *physaddr*

Arguments

physaddr	Use this argument to specify the physical address for which you want to find *tlb* entries.

tlbvtop Command

Use **tlbvtop** to display the R2000/3000/4000 translation buffer entries that map the specified virtual address.

Synopsis

`tlbvtop` *vaddress* `[`*pid*`]`

Arguments

vaddress	Use this argument to specify the virtual address for which you want to find *tlb* entries.
[*pid*]	Use this optional argument to specify the pid associated with the *tlb* entry. If you do not specify a pid, **tlbvtop** defaults to using the *pid* value currently in the coprocessor register **tlbhi**.

unbrk Command

Use **unbrk** to remove breakpoints.

Synopsis

`unbrk` *bpnumlist*

Arguments

bpnumlist	Use this argument to specify the ordinal of a particular breakpoint you want to remove. Use the **brk** command to get the ordinal of a particular breakpoint.

wake Command

Use **wake** to wake up slave processors. **wake** brings all slave processors into a waiting loop, whether those processors are scanning for a keystroke on the console to drop into **symmon** or looking for the address to which to jump to execute the **boot**() routine.

Synopsis

`wake`

wpt Command

Use **wpt** to set a read, write, or read/write watch point on a physical address, using the R4000 watch point registers.

Synopsis

```
wpt [r|w|rw|] [0|physaddr]
```

Arguments

r	Read
w	Write
rw	Read/write
physaddr	Double word aligned address. The watch point will trip on any access within the next eight bytes.
0	An argument of 0 clears the watch point.

Using symmon's Kernel Print Command

The kernel print mode command, **kp**, allows the user to print kernel structures or information summarized from kernel structures. If you do not know the names of the kernel structures or summaries that you want to see, just issue the **kp** command without specifying an argument. If you do not specify an argument for **kp**, the command displays the names of all its subcommands. The **??** command lists the names of all the **kp** subcommands with short descriptions. For systems that do not have **symmon** in PROM, you may omit the leading "kp".

Some of the information you can view using the **kp** command is not of interest to you when debugging device drivers. In the following sections, we describe only those structures and summaries useful when debugging a device driver. **kp** commands that display data are also used with the **idbg** command (without the **kp** keyword).

Note: The following sections mention a number called the "process table entry index." This number is not the process ID. To find the process table entry index that corresponds to a particular process ID, issue the command: **kp plist** *proc_id*, where *proc_id* is the process ID for which you want to find the process table entry index.

kp buf [index/address] – View a buf Structure

The kernel can contain any number of **buf** type structures. Use these structures to manage data transfers to and from a device. To view all of these structures, use **kp buf** without specifying [*index/address*]. To see a particular buf type structure, specify either the buffer index number for the structure or the address for the **buf** type structure as the [*index/address*] argument.

kp eframe address

This displays the exception frame at the given address. The exception frame holds the contents of the general purpose registers at the time the process last executed. If the address is a small integer, the exception frame of the process with that process table index is used.

kp inode number/address – View an inode Structure

Use **kp inode** to view the contents of an **inode** structure. To specify which inode structure you want to see, provide the number or the address of the **inode** structure as the [*number/address*] argument. **kp inode** is obsolete in 5.x. See *vnode* structures.

Synopsis

`kp inode` *number/address*

kp kill addr

Send **SIGKILL** to a process, where *addr* is the process id.

kp mlist

List all dynamically loaded and registered kernel modules.

kp msyms *id*

Use *ID* to print symbols for the dynamically loaded module **ID**, which can be found using the **ml list** command or the **lboot -V** command.

kp pb — Dump Console Print Buffer

This prints the contents of the console print buffer, which can be useful when an important message scrolls off the screen.

kp plist [#] – List Process Summaries

Without an argument, the **kp** command describes all the processes on the system. To see the *plist* information that applies to a particular process, specify the process ID as the [#] argument. Thus, to see the *plist* information that applies to a process with a *pid* of 16672, enter the command:

```
% kp plist 16672
```

The information that *kp plist* [#] displays for a process consists of items such as:

- process table entry number, or index (also known as the process slot number). You use this number, instead of the *pid*, when selecting process-specific information from other kernel structures via **kp**, so be careful not to confuse it with the *pid*.
- process ID
- process status (such as sleep)

kp pda [index] – View a pda Structure

The **pda** structure contains information about a processor's private data area. If you do not specify [*index*], *kp pda* gives you the **pda** structure for all the processors. To see the **pda** structure for a specific processor only, use the [*index*] argument to specify the processor array index for that processor. The value of this argument must be 0 on a single-processor system or a number between 0 and *n* on a multiprocessor system.

kp proc index/address – View a proc Structure

A **proc** structure contains process-specific information not listed in the plist summaries. **kp** requires that you specify the **proc** structure you want to see. You can specify a particular **proc** structure by giving either the process table entry number (*index*) for the process or the address (*address*) for the **proc** structure (in *u.u_proc.p* for the currently mapped process).

kp qbuf device

Dump the contents of buffers queued for the given device. The *device* argument is given as the major/minor device number of the desired device.

kp runq – View the runq Structure

The **runq** structure contains information describing the processes in the run queue. These are processes that are not running but that could run. **kp runq** displays all the information for every process in the **runq** structure.

kp sema address – View a Semaphore Structure

The system uses **sema** type structures to manage the information associated with a semaphore (for example, the semaphore's name, the number of times it was used, whether it is free or locked). Use **kp sema** *address* to list the contents of a particular **sema** structure.

kp slpproc – View Summary of Sleeping Processes

Use this command to view information on all sleeping processes. This information includes such items as the process name, the process address, and the name and address of the semaphore upon which the process is sleeping. **kp slpproc** lists all the information in the *slpproc* structure.

kp ubt [index] – View Subroutine Backtrace Summary

Use this command to view subroutine backtrace information for a user process. This information includes such items as the stack pointer, the caller, and the called function. If you do not specify [**index**], this command reports on the currently mapped user process. To specify a particular user process (one that may or may not currently be mapped) give **kp ubt** [**index**], the index into the process table for the user process in which you are interested.

Caution: If a process is currently running on a CPU, you *must* issue the **ubt** command on the CPU the process is running on. Using **ubt #** on currently running process will produce erroneous results.

kp ubt [*index*] is analogous to the *dbx* command, *where*.

kp user [index] – View a User Structure

The **user** structure contains information that describes a user area. If you do not specify [*index*]**, kp user** displays the user structure for the currently mapped user. To view the user structure for a particular process (one that may or may not currently be mapped), give **kp user** [*index*], the index into the process table for the user process in which you are interested.

kp wd – View SCSI Information

Use **kp WD93** to view information concerning **WD93** SCSI devices.

Synopsis

`kp wd [`*0, 1, 2, address*`]`

Arguments

0	Use this option to display the contents of the WD93 registers. No equivalent yet exists for WD95.
1	Use this option to display the host structure and active *scsi_request_t*.
2	Use this option to show all allocated subchannels.
address	Use this option to show the subchannel at the specified address.

Multiprocessor Debugging

Debugging multiprocessor device drivers with **symmon** is similar to debugging multiprocess code in general. There are two phases: debugging multiprocess code forced to run on a single processor, followed by debugging the same code as it runs on multiple processors.

Configuring the System Software

To force a multiprocess device driver to run on one processor only, you must deactivate all but one processor. Currently, network load is spread across CPUs by default. To preempt network packet processing, see the **rtnetd**(1M) and **network**(1M) man pages.

Preparing the System Hardware

On IRIS-4D and POWER Series systems, multiprocessor debugging requires that you connect an ASCII terminal to each processor. On a two-processor system, the second terminal connects to Serial Port 3. For four-processor systems, the additional terminals connect to ports 5 and 7. Additionally, you must set the MODE switches 4 and 8 (located at the front of the 4D system panel) to "open." When these MODE switches are open, power-on diagnostics display on the terminal attached to each processor. **symmon** loads automatically on all non-console processors, and each prints a message to its terminal. The mode switches are depicted in Table 10-1.

Note: Indigo, Indigo^2, Indy, Crimson, CHALLENGE/Onyx, POWER CHALLENGE/POWER Onyx systems do not require physical manipulation of MODE switches.

symmon and Multiprocessor Debugging

On multiprocessor systems, a separate **symmon** runs on each processor. Breakpoints, however, are shared. When a processor reaches a breakpoint instruction, control is transferred to the **symmon** associated with that processor. Other processors may continue to run independently until they also happen to hit that breakpoint, which can happen if you set a breakpoint in a kernel function or anywhere in a multiprocessor device driver.

At this point, before the kernel is loaded (through a **boot** command from command mode), you can disable processors not associated with the console (the console processor is referred to as the *master processor*) by pressing the **`<Enter>`** key on the keyboard. Disabling processors is useful during the initial debugging phase of a multiprocess program because it is easier to debug a multiprocess program first in single-processor mode. This way, you eliminate the usual sorts of bugs before you have to deal with bugs associated specifically with multiprocessing (such as synchronization bugs).

(See also M. Maekawa, A. Oldehoeft, and R. Oldehoeft. *Operating Systems Advanced Concepts*. The Benjamin/Cummings Publishing Company, Inc., 1987 and A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*, Third Edition. Addison Wesley Publishing Company, 1991.)

By default, all device driver software runs only on the master processor. A driver that understands the multiprocessor environment can indicate that it contains semaphored code through a flag in the master file. However, it is still easier to debug drivers initially on a single-processor system.

If you do not disable the non-master processors, processes run on them as usual. If a breakpoint is encountered on that (non-master) processor, or if a **`<ctrl-A>`** character sequence is entered on that (non-master) processor's terminal, control transfers to **symmon** on that processor. Other processors continue to execute kernel (and user) code until they detect a **`<ctrl-A>`** character sequence, reach a **break** instruction, or hit a lock or semaphore held by the stopped CPU.

Debugging on CHALLENGE/Onyx and POWER CHALLENGE/POWER Onyx systems requires only a single ASCII terminal. All symmon output for all CPUs is multiplexed to the one port. To talk to an individual CPU, use the **cpu#** command. CPUs must be stopped (with the **stop** command) before you can connect to them. To restart all CPUs, use the **C all** command.

Note: It may be necessary to use **`<ctrl-A>`** several times when entering symmon.

Forcing System Memory Dumps

Because the code to force all CPUs, other than the one that does the dump, to spin is at a very high level, the best way to force a memory dump is to use **symmon** to call **panic** with the address of some string in the kernel, as in:

```
kp call panic addr
```

On CHALLENGE/Onyx and POWER CHALLENGE/POWER Onyx series systems, an alternative is to use the NMI button.