Order Number: EK-TCAAD-FS-002

TURBOchannel
Firmware Specification

On-line version.

Digital Equipment Corporation

April 1991

This document was prepared using VAX DOCUMENT, Version 1.2

TURBOchannel Technology Transfer Agreement

To receive free technical support and notice of new TURBOchannel documentation, contact Digital's TRI/ADD Program about free membership at the numbers below.

Grant of Right to Use TURBOchannel Technology

In exchange for your agreeing to the warranty disclaimer and liability limitation stipulated in this Technology Transfer Agreement, Digital Equipment Corporation (Digital) grants at no cost to you a royalty-free nonexclusive license to use TURBOchannel technology (as specified in the TURBOchannel Hardware Specification and the Developer's Kit) to design and develop any kind of option board, computer system, or application-specific integrated circuit (ASIC). This Agreement does not grant you any other rights to Digital's patents, copyrights, trade secrets, trademarks, or licenses to TURBOchannel technology. The purchase cost of the TURBOchannel kit is basically the cost to reproduce the materials.

Warranty Disclaimer

The TURBOchannel technology is transferred "as is." Digital expressly disclaims all implied warranties including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Digital does not warrant, guarantee, or make any representations regarding the use of or the results of the use of the specification and related documents in terms of correctness, accuracy, or reliability. Digital believes the documentation is accurate; however, you must assume the risk as to the results and performance of any product you develop that is based on the TURBOchannel technology.

Limits of Liability

You agree that Digital shall not be liable to you under this Agreement for any damages, including without limitation any lost profits or lost savings, or any consequential, incidental, or punitive damages arising out of the use or inability to use the TURBOchannel Hardware Specification and related documents, or for any claim by another party. Your exclusive remedy under this Agreement shall be the furnishing by Digital of the technical support provided herein. You agree to hold Digital harmless for all claims and damages arising from any third party as a result of their use of or inability to use any product you develop based on TURBOchannel technology.

Digital Equipment Corporation, TRI/ADD Program,
100 Hamilton Avenue, Palo Alto, CA 94301

U.S./Canada	1.800.678.open	Australia	0014.800.125.388
France	05.90.2874	Italy	1678.19087
Japan	0031.12.2363	U.K.	0800.89.2610
Germany	0130.81.1974		

FAX 1.415.853.0155
Internet address: triadd@decwrl.dec.com

Contents

4 System Module Standard Scripts

5 System Module Callback Routines

Figures

Tables

ABOUT THIS SPECIFICATION

Important Changes to the TURBOchannel Firmware Specification. Version 2B of the Developer's Kit introduces these changes to the TURBOchannel Firmware Specification:

- The section ''About This Specification'' now includes a list of changes made in the Firmware Specification.
- *Standard REX Commands*, section **cnfg**, paragraphs 2, 3, and 4 now specify Model 200 of the DECstation/DECsystem 5000 as the source of the cnfg display.

Document Structure. This document contains the specification of TURBOchannel system and option module firmware. System-specific firmware specifications can be found in the system documentation. The first chapter provides a brief overview of TURBOchannel systems. The next three chapters describe TURBOchannel system and option module firmware in detail. Details of the TURBOchannel firmware system module callback routines are documented in Chapter 5.

Conventions Used in This Specification. The Firmware Specification uses these conventions:

- ***Words in bold-face italic type like this*** are terms introduced for the first time.
- **Characters in bold-face type like this** are characters that you type exactly as they are printed.
- *Terms in italic type like this* are variables that you replace with actual values.
- The number sign (#) represents slot numbers, which are numbers that the firmware uses to identify TURBOchannel modules.
- Square brackets ([ ]) surround optional arguments.
- Parentheses surround groups of terms from which you select only one term. Vertical bars (|) separate the terms from which you can choose.
- Ellipses (...) follow an argument that can be repeated.

Call Prototype. This document defines all program call interfaces in ANSI C. For example:

```
char *strncat(char *s1, char *s2, int n);
```

defines the call interface to the **strncat** routine; it is called a ***call prototype***. ANSI C conventions are used to indicate parameter and routine type. Where defined types are used, a **typedef** statement defining the type appears before the call prototype.

Hex Offsets. Throughout this document, field offsets of data structures are shown in hexadecimal format unless otherwise indicated.

1. TURBOCHANNEL SYSTEM OVERVIEW

TURBOchannel is a low cost, high performance module interconnection technology based on hardware, software, and ***firmware*** components. The firmware components are specified in this document, hardware and software components are specified elsewhere.

TURBOchannel Modules. At the core of the TURBOchannel hardware is a synchronous, asymmetrical I/O channel used to connect ***option modules*** to a ***system module***. The channel is asymmetrical in that the system module has read or write access to an option module and an option module has read or write access to the system module, but option modules have no access to other option modules.

Figure 1 shows permitted access between TURBOchannel system modules and option modules.

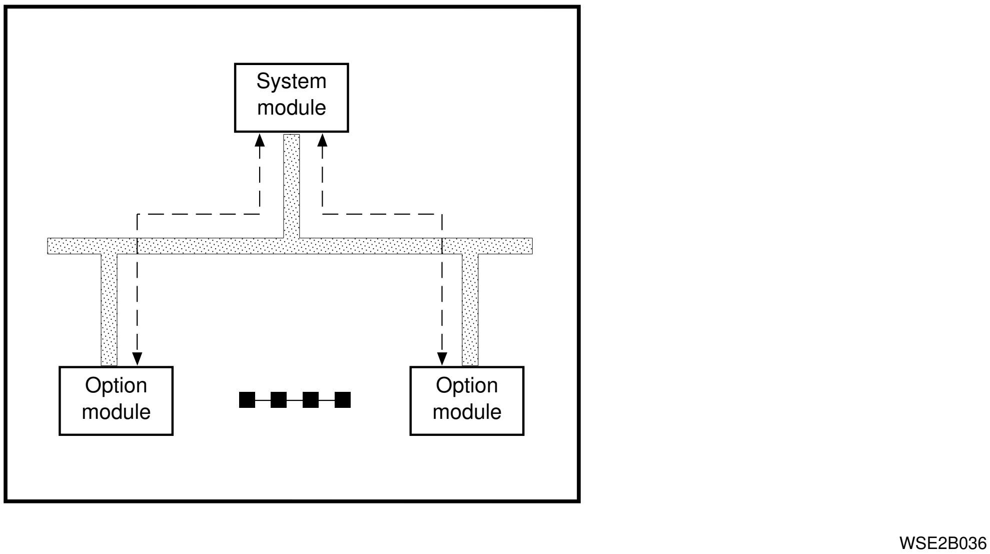

Figure 1. TURBOchannel I/O access

TURBOchannel Slots. TURBOchannel systems divide a part of the system physical address space into ***slots*** where all the addresses within a slot are assigned to a TURBOchannel module. One slot is always reserved for the system module. The base address of the first slot, the size of each slot, and the number of slots are system-specific parameters.

Figure 2 shows how physical address space in a TURBOchannel system is divided into slots.

The slot number of the system module is fixed for any implementation and is typically assigned the highest slot number. The slot number of an option module depends on which option connector it is connected to. Some systems may implement fewer option module connectors than there are slots. The missing slots are either unusable or they can be used for ***integral options*** (components that appear to be separate TURBOchannel options but instead are part of the system module).

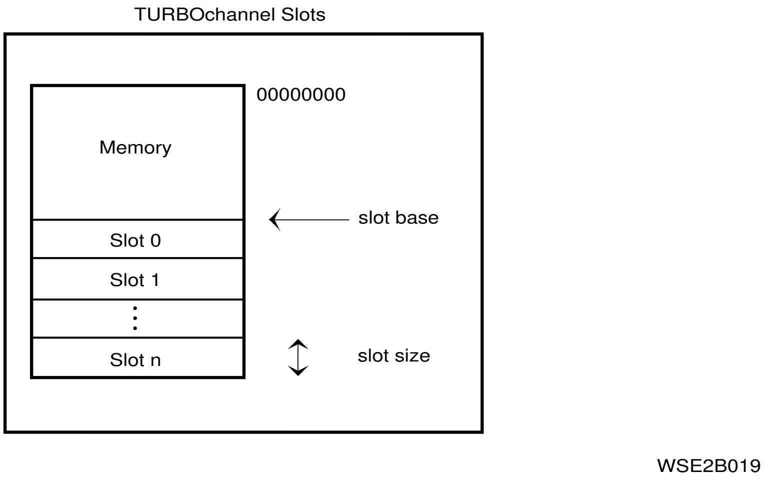

Figure 2. TURBOchannel slot address space

TURBOchannel Module ROMs. A system module contains a ROM that contains system module firmware. An option module contains a ROM that contains information about the option module, the ROM geometry, and it can also contain option module firmware.

The ROM Executive. The system module firmware includes a ***ROM Executive Program*** (REX) that is responsible for controlling the system. REX commands are used to invoke diagnostics and initialization routines, to boot the system, to examine and modify memory, and to perform other system management functions. You enter REX commands at the system console or they are read from ***scripts*** stored in ROM.

ROM Objects. REX makes extensive use of ***ROM Objects***. ROM objects are named containers for module-specific scripts and firmware that are stored in ROM. Objects are part of the mechanism that REX uses to perform module dependent operations such as ***bootstraps*** and diagnostics. For example, to boot a program from a disk controller module, REX loads the object named "boot" into memory from the controller module and then calls it to load the program from the disk. By performing all module-specific functions using firmware or scripts from the modules themselves, REX remains device independent and new modules can be easily developed and added to TURBOchannel systems.

Diagnostics. REX invokes all system- and option-module diagnostics. Diagnostic routines are module-specific and are stored in a test object. REX loads the test object into memory and calls it to perform module diagnostics.[1] Diagnostics can be divided into small segments that are invoked to test portions of a module. These tests are invoked by name. When a test fails, the module number and the test name are displayed and can be used to quickly identify the failing unit.

REX uses standard scripts from each module to invoke diagnostics. When REX is invoked after power is applied to the system, it automatically invokes the system-module script named "powerup". The power-up script invokes the power-up self-test scripts, named "pst-t" or "pst-q" for thorough or quick testing respectively, of the system module and then of each option module. These scripts invoke the appropriate power-up tests. Additional nonpower-up diagnostic scripts can be included in module ROMs and can be invoked manually to perform special tests.

[1] Some system module diagnostics are not implemented in this manner. Diagnostics that test system resources required by REX are executed directly from the system-module ROM before REX is invoked.

Bootstraps. Bootstraps are invoked by REX. A bootstrap loads a program into memory from a boot module. A disk controller module or an Ethernet controller module are typical boot modules. To perform a bootstrap, REX loads the object named "boot" from the boot module and then calls it to load the program. REX transfers control to the program after it is loaded.

System Console. REX uses the system console to display status messages and to accept operator commands. The system console can be either a terminal or a combination of a CRT display and a keyboard device. Modules that act as console devices contain ***console drivers*** that REX uses to communicate with the device. Console device assignments are made either by the operator or by a system-specific ***autoconfigure*** procedure. The assignments are stored in nonvolatile memory that is part of the system module.

2. SYSTEM MODULE FIRMWARE

The system module is the core of a TURBOchannel system. The TURBOchannel firmware architecture imposes few constraints on system module implementation. The imposed constraints are related to the interface between system module firmware and option module firmware (discussed in the next chapter).

Control passes to the system-module firmware whenever a hardware reset occurs or when software explicitly branches to it (a case that is treated the same as a hardware reset). A hardware reset occurs when power is applied and can also occur in response to a major system error or operator action.

The routine that receives control on reset is the ROM Executive (REX). When invoked, REX determines if the reset is the result of power being applied. If so, REX performs system module power-up testing and then invokes all module-specific power-up tests. Following completion of power-up testing, system- and option module-specific initialization is performed. If the reset is not due to power-up, only the initialization is performed.

REX implements a small set of standard commands that can be entered either at the console terminal or from a script. Option-module diagnostics are performed by reading standard diagnostic scripts that contain commands to invoke module-specific diagnostics.

REX is also responsible for configuring a device or pair of devices as the console terminal. After the console terminal is initialized, error messages can be displayed and operator commands can be entered.

Standard REX Commands. Standard REX commands are responsible for bootstrapping the system, displaying system configuration information, initializing the system, and performing system diagnostics. These commands cause TURBOchannel module-specific firmware to be executed. These commands are accepted either from the system console or from scripts.

All standard REX commands are described below and include **boot**, **cnfg**, **init**, and **t**. To use the standard commands, type the command as shown in the description, add the appropriate options and arguments, and press Return.

The prompt (>>) is shown on the command line for clarity.

boot.

BOOT >>**boot** [[*OPTIONS*] *#/path* [*argument...*]]

OPTIONS	**-n**	load but do not execute
	-z *number*	sleep for *number* seconds

The **boot** command loads and optionally executes the program specified by *#/path*, where # is the slot number of the module acting as the boot device and *path* is a device-specific file specification. The **-n** option suppresses execution of the program after it is loaded. The **-z** option causes the system to wait for *number* seconds before starting the bootstrap. If no arguments are specified, the contents of the **boot** environment variable are used as the argument list.[1]

For example, you can use the **boot** command to specify the VMUNIX file on a disk drive whose SCSI ID is 0 and is part of the SCSI bus connected to option slot 5 by typing **boot 5/rz0/vmunix**.

[1] A system-specific command may be used to set the value of the **boot** environment variable.

cnfg.

CNFG **cnfg** [*#*]

The **cnfg** command displays system configuration information. If you specify a module number (*#*), detailed configuration information for the module is displayed, otherwise brief configuration information for each module in the system is displayed.

This is an example of a DECstation 5000 Model 200 **cnfg** display with no slot number specified. This display shows a system with optional Ethernet, SCSI, and color frame buffer modules:

```
>>cnfg
 7:  KN02-AA  DEC  T5.2a  TCF0  ( 24 MB)
 6:  PMAD-AA  DEC  T5.2a  TCF0  (enet: 08-00-2b-0c-e0-d1)
 5:  PMAZ-AA  DEC  T5.2a  TCF0  (scsi = 7)
 2:  PMAD-AA  DEC  X5.1f  TCF0  (enet: 08-00-2b-0f-43-31)
 1:  PMAZ-AA  DEC  X5.1e  TCF0  (scsi = 7)
 0:  PMAG-BA  DEC  T5.2a  TCF0  (CX -- d=8)
```

This is an example display of configuration information for the system and memory modules in slot number 7 of a DECstation 5000 Model 200:

```
>>cnfg 7
 7: KN02-AA  DEC     T5.2f    TCF0    ( 24 MB)
        mem( 0): a0000000:a07fffff    (  8 MB)
        mem( 1): a0800000:a0ffffff    (  8 MB)
        mem( 2): a1000000:a17fffff    (  8 MB)
```

The contents of a detailed configuration information display is module-specific. For example, the DECstation 5000 Model 200 SCSI controller module, which is in slot 5, generates a **cnfg** display similar to this:

```
>>cnfg 5
 5: PMAZ-AA  DEC     T5.2e    TCF0     (SCSI = 7)
    ----------------------------------------------------
    DEV   PID                VID          REV     SCSI DEV
    ===== ================= ============ ======= =========
    rz0   RZ55     (C) DEC  DEC          0700    DIR
    rz1   RZ56     (C) DEC  DEC          0200    DIR
    tz3                                          SEQ
```

init.

INIT >>**init** [#] [*arguments...*]

The **init** command initializes module hardware. If you specify a module number (**#**), only that module is initialized, otherwise all modules are initialized.

The interpretation of arguments is module-specific.

t.

TEST >>**t** [*OPTIONS*] *#/testname* [*argument...*]

OPTIONS **-l** Loop

The **t** command runs module tests. A test is specified by *#/testname*, where *#* is the slot number and *testname* is a module-specific test name.

The **-l** option causes a test to be executed continuously until a system reset occurs or Ctrl-c is pressed.

System-Specific REX Commands. Additional system-specific REX commands are both allowed and expected. Examples are commands that allow the operator to examine and modify memory, assign values to REX environment variables, and implement security features.

System-specific commands cannot be contained in option module scripts, but they can be contained in system-module scripts.

REX Environment Variables. REX uses environment variables to store system parameters and to pass information to the operating system. Some environment variables are normally retained in nonvolatile memory, others are lost when REX exits.

When reading commands from scripts, REX performs environment variable substitution. If the string **$X** or **${X}** appears in a script, REX replaces it with the contents of the environment variable **X**. REX uses double quotes (") and single quotes (') as quote delimiters. If **$X** or **${X}** appear between double quotes, environment variable substitution is performed, if they appear between single quotes, substitution is not performed.

Standard Environment Variables. The following standard environment variables are defined:

boot	*[1]	Specifies the default arguments for the boot command.
console	*	Controls the choice of the system console. If any value other than "s", the system autoconfigures the console.[2]. If "s", the system uses a terminal connected to a system module. That is, setting **console** causes the system to immediately reconfigure and initialize the system console.
haltaction	*	Specifies system halt actions: "b" causes the system to perform a boot command, "h" causes the system to halt (accept command from the console), "r" causes it to perform a restart command and if that fails, perform a boot command.
more	*	Contains the screen height in lines. If nonzero, the system paginates all command output using the value as the page size.
osconsole		Contains the slot numbers of the console driver(s). If a tty driver from slot x is used as the console, **osconsole** is set to x. If a CRT driver from slot *y* and a keyboard driver from slot *z* are used as the console, **osconsole** is set to *y,z*.
testaction	*	Controls the power-up self-test behavior: "t" specifies a thorough (but slower) testing of the system. "q" specifies a quicker but less thorough testing of the system. "m" specifies manufacturing-specific tests be performed.[3]
#		Contains the module number of the module containing the current script. If no script is active, the base system module is specified.

[1]Environment variables marked with * are preserved in nonvolatile RAM.

[2]The autoconfigure process is system-specific.

[3]A ***testaction*** value of "m" cannot be set by the operator. It is only set if a system-specific manufacturing "jumper" is installed. The jumper overrides any other setting stored in nonvolatile RAM.

Additional environment variables may be set as implementation-specific side effects of various bootstrap and test procedures.

REX Memory Regions. REX executes directly from the system module ROM. However, REX uses portions of system RAM to execute module-specific firmware. REX divides memory into 4 regions referred to as memory regions 0, 1, 2, and 3.

Table 1 lists the addresses and use of each memory region in REX.

Table 1. REX Memory Regions

Region	Starting Address	Ending Address	Use
0	0xa0000000	0xa000ffff	restart block, exception vectors, REX stack and bss
1	0xa0010000	0xa0017fff	tty or keyboard drivers
2	0xa0018000	0xa001f3ff [1]	CRT driver
3	0xa0020000	0xa002ffff	boot, cnfg, init, and t objects

[1]Note that the last 3Kb of region 2 is reserved for backward compatibility with previous system software.

When REX loads firmware into a region, it first zeros the entire region. However, REX may record what firmware was last loaded and not reload already loaded firmware. Caution must be taken in assuming that any part of a region is zeroed.

Note that the addresses are MIPS KSEG1 addresses. Code executing from these addresses is not cached.

Program Interface. When REX transfers control to a program in memory, either as a result of a **boot** command or some other system-specific command, REX calls the program with the following interface:

```
int program(int argc, char **argv, int magic, rcv *vector)
```

where **argc** and **argv** have the standard meanings, **magic** is 0x30464354, and **vector** is a pointer to the callback vector.

3. OPTION MODULE FIRMWARE

Option module firmware contains module-specific routines used for booting, displaying configuration information, performing console input and output, performing module-specific initialization, and performing module-specific tests. All option module firmware is stored in objects in the option module ROM.

Scripts are also stored in objects in option module ROMs. Scripts are used to organize diagnostic test sequences.

Option Module ROM Format. All option modules include a ROM located at the base of the TURBOchannel address space.[1] The ROM contains a header section and a ROM objects section. The ROM header section begins at offset 0x000, and the ROM objects section begins at offset 0x480.

ROM Header. The ROM header must be present in the option module ROM if the system is to recognize the module.[2] The ROM header contains information used by the system module firmware (and the operating system) to determine if a module is present, what type of module is present, and critical ROM geometry information.

Figure 3 shows the assignment of fields in the ROM header.

---

[1] Some older TURBOchannel options placed their ROM at offset 0x3C0000 from the base. System configuration code must check first at the old address, then at the new address.

[2] The ROM header should not be accessed for the first 200 milliseconds following system reset. For some older TURBOchannel options, the ROM header does not appear valid during this period.

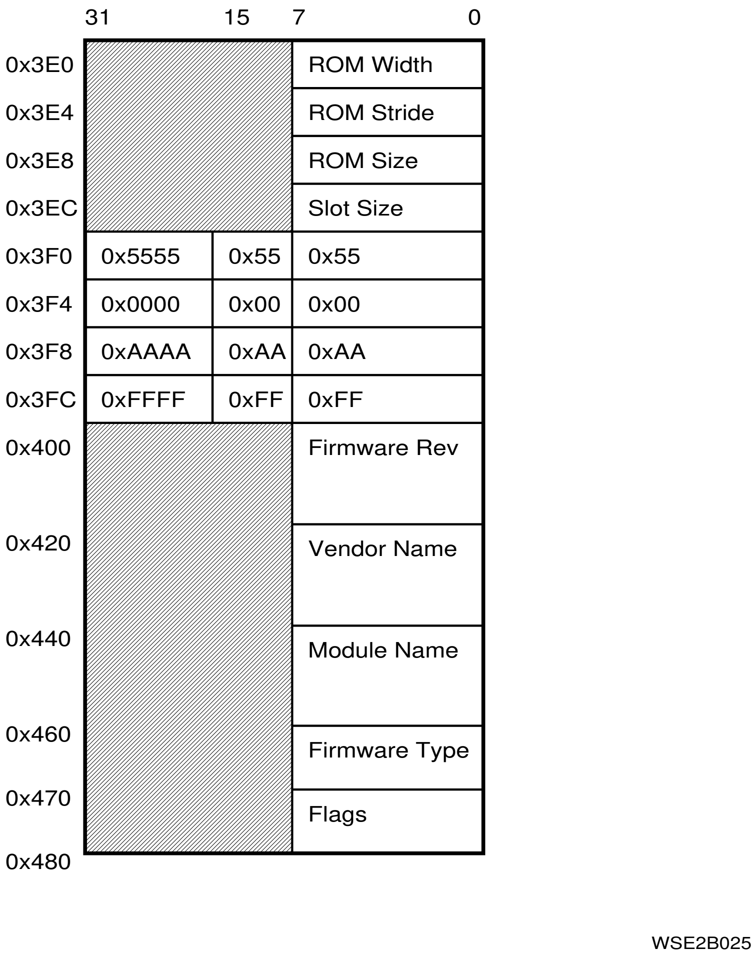

Figure 3. ROM header locations

Note that only the first byte of each word of the ROM header contains information.

The first 992 (decimal) bytes of the ROM header are reserved for use by the option module.

The **ROM width**, **ROM stride**, and **ROM size** fields contain ROM geometry information. **ROM width** is the width of the ROM in bytes. It can be 1, 2 or 4. **ROM stride** is the address stride of the ROM as seen by the system module; it must be 4. **ROM size** is the size of the ROM divided by 8192.

The **slot size** field contains the minimum TURBOchannel slot size required by the option module divided by 4,194,304 (4-Megabytes). For example, the DECstation 5000 **slot size** is 1.

Four **pattern** fields are present in the ROM header: **55**, **00**, **AA** and **FF**. These patterns are used for test purposes and as part of the ***signature*** of a valid ROM header.

The **module version** field contains 8 ASCII characters of version information. Only graphic ASCII codes (i.e., values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.

The **module vendor** field contains 8 ASCII characters indicating the module manufacturer. Only graphic ASCII codes (i.e., values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.

The **module name** field contains 8 ASCII characters indicating the module name. Only graphic ASCII codes (i.e., values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.

The **firmware type** field contains 4 ASCII characters indicating the type of firmware present in the module ROM. Only graphic ASCII codes (i.e., values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks. All modules conforming to this specification must contain the ASCII characters "TCF0" (TCFzero) in this field.

The **flags** field is a 4-byte field with bit<0> indicating whether the module implements parity. When the bit is one, the option module implements TURBOchannel parity. When the bit is zero, the option module does not implement TURBOchannel parity. All remaining bits are reserved and should be zero.

ROM Objects. The presence of the ROM objects section of the option module ROM is optional. A ROM object contains both header information and an object body. REX uses the header information to locate objects, to determine the object type, and determine where to load firmware objects into RAM. The body contains either code or text.

Figure 4 shows how fields are assigned in the ROM object section.

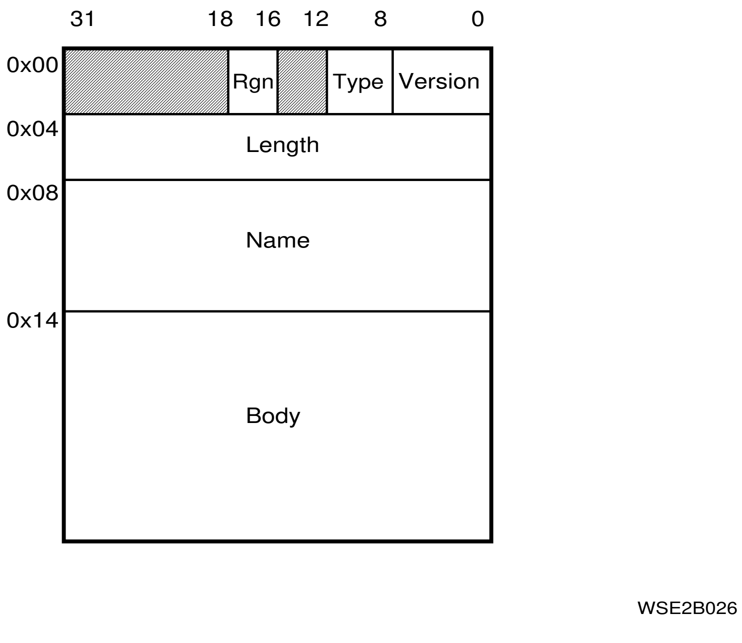

Figure 4. ROM object fields

The **version** field is an 8-bit field containing the ROM object version number.

The **type** field is a 4-bit field which contains a code indicating the object body type. Valid **type** values are 0, 4, and 5:

0 MIPS I code. The body of this type of object contains executable code. This code must contain only instructions from the MIPS I instruction set compiled for little-endian byte order. Coprocessor instructions must not be present.

4 ASCII text. The body of this type of object contains a zero terminated ASCII string stored in little-endian byte order.

5 Symbolic Link. The body of this type of object contains a zero terminated ASCII string stored in little-endian byte order. The contents of the string is the name of another object. Symbolic links can be used to redirect a reference from one object name to another. This redirection is only allowed between objects in one module, redirection to another module is not allowed.

All other values for type are reserved.

The **rgn** field is a 2-bit field that specifies the region into which a code object is loaded for execution. Valid values are 1, 2, and 3. See the section on "*REX Memory Regions*".

The **length** field is a 32-bit field containing the length in bytes of the object including the header information. The length must be a multiple of 4.

The **name** field is a 12-byte field that contains the name of the object in ASCII. Only graphic ASCII codes (i.e., values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.

The **body** portion of the object is the object content. The length of **body** must be a multiple of 4.

Multiple ROM objects are stored in the option module ROM beginning at offset **0x480**. Gaps are not allowed between objects. A special object with a length of zero terminates the ROM objects section of the ROM.

Option Module Firmware. All option module firmware is contained in one or more ROM objects stored in the module ROM. Option module firmware is not executed directly from the option module ROM, it is always copied to RAM for execution.

All option module firmware is called at the base of the memory region where it is loaded. The call interface is identical for all firmware:

```
int  function(int argc, char **argv, int slot, rcv *vector);
```

where **argc** and **argv** have their standard meanings, **slot** is the slot number of the module and **vector** is a pointer to the callback vector.

There are seven standard ROM executable objects: "kbd", "tty", "crt", "boot", "cnfg", "init", and "t". The first three implement the console terminal interface, the remaining implement the REX **boot**, **cnfg**, **init**, and **t** commands respectively.

The "kbd" and "tty" objects must have 1 specified in the **rgn** fields. The "crt" object must have 2 specified in the **rgn** field. All remaining objects must have 3 specified in the **rgn** fields. Each object must be linked to execute in the region specified.

The **initc**, **getc**, and **putc** functions are only supported by "kbd", "tty", and "crt" objects. These are console driver objects. When REX configures the console terminal interface, it loads the appropriate driver objects into memory for use. REX calls the **initc** function of the object once before calling the **getc** or **putc** functions. If REX reinitializes the console for any reason, the call sequence is repeated but the driver objects are not reloaded. Console I/O is restricted to 8-bit characters using the ISO-Latin-1 character set.

When REX executes a **boot** command, it first loads the "boot" object and then calls the **boot** function to perform the boot. The same procedure is followed when REX executes a **cnfg** command, an **init** command, and a **t** command. Note however, a single object can implement more than one of these functions by redirecting the object references using symbolic links and by dispatching to functions using ***argv[0]** as the key.

Option module firmware can call system module routines via the ROM callback vector. These routines are documented in Chapter 5.

Standard option module routines are described below.

boot.

```
int  boot(int argc, char **argv, int slot, rcv *vector);
```

This function loads a program into RAM. It returns the execution address of the program if the load succeeds, otherwise it returns 0.

The REX **boot** command loads the object named "boot" from the specified module ROM and calls it to load a program. If no object named "boot" is found, the **boot** command fails.

The value of ***argv[0]** is "boot". The value of ***argv[1]** is a string with the form *#/path*. The interpretation of *path* and any remaining **argv** entries is module-specific.

The boot object must also contain a **bootinit**, **bootread**, and **bootwrite** routine.[1] The boot object must store the addresses of these routines into the callback vector. When REX invokes the program, it passes the address of the callback vector to it. By placing the addresses of the **bootinit**, **bootread**, and **bootwrite** routines in the callback vector, they are made available to the program. The specifications of these routines are in Chapter 5.

cnfg.

```
int cnfg(int argc,char **argv, int slot, rcv *vector);
```

This function displays configuration data about a module. It returns 0 if successful, otherwise it returns a negative value.

The REX **cnfg** command loads the object named "cnfg" from the specified module ROM and calls it to display module-specific configuration information. If no object named "cnfg" is found, it is not an error condition, but REX only displays information available from the ROM header of the module.

The value of **argc** can be 1 or 2. The value ***argv[0]** is "cnfg". If **argc** is 2, ***argv[1]** is a string containing the module number.

If **argc** is 1, **cnfg** displays a brief, possibly null, configuration report. A brief display should contain 30 or fewer characters and must contain printing characters only.

[1] While not required, boot routines may interpret a **"-Noboot"** argument as a request to perform all the initialization needed to use the **bootinit**, **bootread**, and **bootwrite** routines, but not to attempt loading of any program that is specified. The **"-Noboot"** option should cause the boot routine to return 0 as status. Existing software has used this as part of a crash-dump mechanism where the crash-dump code issues a boot request with **"-Noboot"** specified to load the boot routine and then use the **bootinit**, **bootread**, and **bootwrite** routines to dump the contents of memory.

This example shows a brief **cnfg** report from each module in a DECstation/DECsystem 5000 Model 200 that has three TURBOchannel option modules. The text inside parentheses at the end of each line is displayed by the **cnfg** routine of each module. The remaining text is displayed by REX.

```
>>cnfg
 7:  KN02-AA  DEC  T5.2a  TCF0  ( 24 MB)
 6:  PMAD-AA  DEC  T5.2a  TCF0  (enet: 08-00-2b-0c-e0-d1)
 5:  PMAZ-AA  DEC  T5.2a  TCF0  (scsi = 7)
 2:  PMAD-AA  DEC  X5.1f  TCF0  (enet: 08-00-2b-0f-43-31)
 1:  PMAZ-AA  DEC  X5.1e  TCF0  (scsi = 7)
 0:  PMAG-BA  DEC  T5.2a  TCF0  (CX -- d=8)
```

If **argc** is 2, **cnfg** routine can display a more detailed configuration report. Each line is terminated with a NL character.

This example shows a detailed report from module 7 in a DECstation/DECsystem 5000 Model 200. The text in parentheses in the first line and each line that starts with *mem* is displayed by the **cnfg** routine. The remaining text is displayed by REX.

```
>>cnfg 7
 7: KN02-AA  DEC     T5.2f    TCF0   ( 24 MB)
        mem( 0): a0000000:a07fffff   (  8 MB)
        mem( 1): a0800000:a0ffffff   (  8 MB)
        mem( 2): a1000000:a17fffff   (  8 MB)
```

getc.

```
int  getc(int argc, char **argv, int slot, rcv *vector);
```

This console driver function is called to input a single character. The value of **argc** is 1 and the value of ***argv[0]** is "getc". If an input character **c** is available, it is returned as **(int)(unsigned char)c**. If input is not available, 0 is returned. If a break condition is detected, a negative value is returned. Input is not echoed.

init.

```
int init(int argc, char **argv, int slot, rcv *vector);
```

This function initializes a module.

The REX **init** command loads the object named "init" into memory and calls it to perform module-specific initialization. If no object named "init" is found in the option module ROM, it is not an error, no module-specific initialization is performed.

The value of **argc** can be 1 or greater. The value of ***argv[0]** is "init". If **argc** is greater than 1, the value of ***argv[1]** is a character string containing the module slot number. The interpretation of remaining **argv** entries is module-specific. This function returns 0 if it is successful, otherwise it returns a negative value.

The execution of the **init** routine by a module used as a console interface may erase the screen of a console device but must not require reinitialization of the console driver.

initc.

```
int initc(int argc, char **argv, int slot, rcv *vector)
```

This console driver function initializes the console driver, it is called before either **getc** or **putc** are called. It returns 0 if successful, otherwise it returns a negative value.

After REX loads console drivers, it calls **initc** to initialize them. **initc** is called at least once before **getc** or **putc** are called. The value of **argc** is 1 and ***argv[0]** is "initc".

putc.

```
int putc(int argc, char **argv, int slot, rcv *vector);
```

This console driver function is called to output a single character. The value of **argc** is 2, the value of ***argv[0]** is "putc", and the value of ***argv[1]** is the address of the character to be output. If output is possible, the character is output and 0 is returned, otherwise a negative value is returned.

t.

```
int t(int argc,char **argv,int slot, rcv *vector);
```

This function invokes a module-specific test. It returns 0 if successful, otherwise it returns a negative value:

-1 if the test failed for a reason that is unlikely to affect system operation
-2 if the test failed for a reason that is likely to affect system operation
-3 if the test failed for a reason indicating a severe problem, a problem likely to affect the operation of REX.

The REX **t** command loads the object named "t" into memory and calls it to perform module-specific tests. If no test object is found, the **t** command fails.

The value of **argc** can be 2 or greater. The value of ***argv[0]** is "t". The value of ***argv[1]** is a string of the form *#/testname*, where *#* is the module number and *testname* is the name of the module-specific test to be performed. The interpretation of remaining **argv** entries is test-specific.

4. SYSTEM MODULE STANDARD SCRIPTS

REX uses three standard diagnostic scripts to invoke module-specific diagnostics: **pst-t**, **pst-q**, and **cnsltest**. REX invokes these scripts automatically when power is applied to the system. Additional scripts can be present that the operator invokes from the console terminal. Option module scripts must contain only REX **t** commands.[1] All of these scripts are optional. If not present in a module, power-up testing of the module is not performed.

pst-t. Executed on power-up when the **testaction** environment variable is set to "t", **pst-t** script invokes thorough module diagnostics. For example:

```
t ${#}/regs
t ${#}/ram
t ${#}/esar
t ${#}/int-lb
t ${#}/ext-lb
t ${#}/crc
t ${#}/cllsn
t ${#}/promisc
t ${#}/m-cst
t ${#}/int
t ${#}/reg
```

This script invokes 10 diagnostic tests. Note that the TURBOchannel slot number is specified by **${#}**. During processing, REX replaces this with the value of the **#** environment variable which always contains the TURBOchannel slot number of the currently executing script. This technique is required as a module may be plugged into any slot.

pst-q. Executed on power-up when the **testaction** environment variable is set to "q", the **pst-q** script also invokes module diagnostics. It is similar to the **pst-t** script except that it is quicker; either fewer tests are executed, quicker tests are executed, or both. For example:

```
t ${#}/regs
t ${#}/esar
t ${#}/crc
t ${#}/cllsn
t ${#}/promisc
t ${#}/m-cst
t ${#}/int
```

Note that in comparison to the previous script, several tests have been removed, presumably tests that take a long period of time to execute. A reasonable goal is that a **pst-q** script should complete execution in less than six seconds.

cnsltest. Executed on power-up before the console terminal is enabled, the **cnsltest** script invokes tests that would disrupt the normal operation of the console. Only modules that contain console drivers need to provide this script.

[1] System module scripts can contain any standard REX command as well as any system-specific command.

5. SYSTEM MODULE CALLBACK ROUTINES

The system module implements a set of routines that are available to option module ROM firmware via a ***call vector***. The address of the call vector is passed to all ROM object firmware and to programs loaded by the **boot** command. The routines are provided for general utility or to isolate option module firmware from system dependent features. All calls are performed using MIPS calling conventions. The following terms are defined: **NULL**(0), **EOF**(-1), and **NL**(0xA). All characters strings are in little-endian order.

Callback Routine Descriptions. The callback routines described below are followed by a table of callback vectors.

bootinit.

```
int bootinit(void);
```

This routine is called to initialize the bootstrap input/output routines **bootread** and **bootwrite**. This routine must be called once before using **bootread** or **bootwrite**. This routine returns 0 if successful, otherwise it returns a negative value.

bootread.

```
int bootread(int b, void *buffer, int n);
```

This function reads **n** bytes using the boot driver into the array of bytes with **buffer** as the start address. If the boot device is block structured, the read begins at block address **b** (blocks are assumed to be 512 bytes long), otherwise the interpretation of **b** is device-specific. The routine returns the number of bytes read or a negative value if an error is detected. (See "*Option Module Firmware*" in Chapter 3 for more information on this routine.)

bootwrite.

```
int bootwrite(int b, void *buffer, int n);
```

This function writes **n** bytes using the boot driver from the array of bytes with **buffer** as the start address. If the boot device is block structured, the write begins at block address **b** (blocks are assumed to be 512 bytes long), otherwise the interpretation of **b** is device-specific. The routine returns the number of bytes written or a negative value if an error is detected. (See "*Option Module Firmware*" in Chapter 3 for more information on this routine.)

clear_cache.

```
void clear_cache(void);
```

This routine clears all entries from the processor cache (if any).

console_init.

```
int console_init(void)
```

This routine initializes the console I/O routines. This function returns 0 if successful, otherwise it returns a negative value.

disableintr.

```
int disableintr(int sn);
```

This function disables interrupts from the TURBOchannel option module in slot **sn**. This function returns 0 if successful, otherwise it returns a negative value.

enableintr.

```
int enableintr(int sn);
```

This function enables interrupts from the TURBOchannel option module in slot **sn**. Once the interrupt is enabled, **testintr** can be used to determine if an interrupt is pending. This function returns 0 if successful, otherwise it returns a negative value.

execute_cmd.

```
int execute_cmd(char *cmd);
```

This function requests REX to execute the command string whose address is in **cmd**. This function returns 0 if successful, otherwise it returns a negative value. This function should not be called by option module firmware, it is only to be called by system software.

getbitmap.

```
     typedef struct{ int pagesize; unsigned char bitmap[];}
memmap;
     int getbitmap(memmap *map);
```

This function is called to construct a map of available memory at the address specified in **map**. It returns the number of bytes in **bitmap**. If **map** equals **NULL**, no map is constructed but the number of bytes in the map is still returned.

The **pagesize** field specifies the memory system page size in bytes. The value of **pagesize** can be different than that normally used by the operating system.

Bitmap is a byte array. If bit_i of $byte_j$ is 1, $page_{8j+i}$ is available. If it is 0, the page is unavailable. A page is unavailable if it does not exist, is unreliable, or is reserved. Operating system software must not use pages marked as unavailable.

getchar.

```
int getchar(void);
```

This function reads the next input character **c** from the console and returns **(int)(unsigned char)c**. If an end-of-file condition (Ctrl-d) is detected, it returns **EOF**. Input characters are not echoed.

getenv.

```
char *getenv(char *name);
```

If the environment variable **name** is defined, this function returns a pointer to its value, otherwise it returns **NULL**. The pointer points to a temporary string which may be destroyed by subsequent calls to **setenv** or **getenv**.

gets.

```
char *gets(char *s);
```

This function reads characters from the console and stores them in the string **s** until a **NL** character is stored or an end-of-file condition (Ctrl-d) occurs. If any characters are stored in **s**, this function returns **s**, otherwise it returns **NULL**. No means are provided to limit the number of characters that this routine reads and stores. Characters are echoed as they are read and system-specific line editing functions may be performed.

getsysid.

```
int getsysid(void);
```

This function returns a value containing system information. Bits 0:7 contain the hardware version level, bits 8:15 contain the firmware revision level, bits 16:23 contain the system type, and bits 24:31 contain the processor type.

Values for each field are specified by the system parameter specification for each system. For example, a DECstation 5000 Model 200 returns the following:

```
0x 82 02 02 20
   __ __ __ __
   |  |  |  |____ rev 20 of the R3000
   |  |  |_______ TCF0 firmware
   |  |__________ DECstation 5000
   |_____________ MIPS R3000
```

gettcinfo.

```
typedef struct{
  int revision;
  int clk_period;
  int slot_size;
  int io_timeout;
  int dma_range;
  int max_dma_burst;
  int parity;
  int reserved[4]
} tcinfo;

tcinfo *gettcinfo(void);
```

This function returns a pointer to a structure containing TURBOchannel implementation parameters. Table 2 lists the TURBOchannel parameters in this structure and gives sample parameters from a DECstation/DECsystem 5000 Model 200:

Table 2. gettcinfo Implementation Parameters

Parameter	Description	DS5000 Example
revision	Hardware revision level	1
clk_period	Clock period in nanoseconds	40
slot_size	Slot size in megabytes	4
io_timeout	I/O timeout in cycles	255
dma_range	DMA address range in megabytes	480
max_dma_burst	Maximum DMA burst length	128
parity	True if system module supports TURBOchannel parity	0

Remaining fields are reserved for future use.

halt.

```
void halt(int* v, int cnt);
```

This function is called to "halt" the system. The **cnt** values whose first address is **v** are printed on the console and then REX is initialized and commands are accepted from the console terminal. Return from REX to the caller can be provided by a system-specific REX command.

io_poll.

```
int io_poll(void);
```

This function is called to allow REX to check for pending console input or output. This function raises **SIGINT (4)** if Ctrl-c is typed. In addition, it enables or disables all console output respectively if Ctrl-q or Ctrl-s is typed. This function returns a nonzero value if any console input is pending, otherwise it returns zero.

leds.

```
void leds(int value);
```

This routine causes **value** to be displayed on the system module ***LED*** display. The number of bits of **value** that can be displayed is system dependent and can be zero. This routine should be used by option modules only for debugging purposes, not for error reporting.

longjmp.

```
typedef  int jmp_buf[12];
void longjmp(jmp_buf env, int value);
```

This routine causes a second return from the **setjmp** routine that stored its context in **env**. If **value** is not zero, **value** is returned by **setjmp**, otherwise 1 is returned.

memcpy.

```
void *memcpy(void *s1, void *s2, int n);
```

This function copies the sequence of **n** bytes beginning at address **s2** to the address beginning at address **s1**. It returns **s1**.

memset.

```
void *memset(void *s1, int c, int n);
```

This function sets the sequence of **n** bytes beginning at address **s1** to the value **c**. It returns **s1**.

msdelay.

```
void msdelay(int delay);
```

This routine waits for **delay** milliseconds to elapse before returning. The accuracy of this routine is system dependent.

puts.

```
int  puts(char *s);
```

This function writes the string **s** to the console. It writes a **NL** character in place of the terminating **NULL** character in **s**. If no error is detected, it returns 0; otherwise it returns **EOF**.

printf.

```
int printf(char *format, ...);
```

This function generates formatted output to the console. It returns the number of characters generated, or it returns a negative value if an output error occurs. The format functions are a subset of those defined for ANSI C:

FMT	*% [FLAG] [WIDTH] FORMAT*
FLAG	- | + | *0*
WIDTH	*number*
FORMAT	*x* | *X* | *d* | *o* | *u* | *c* | *s*

raise.

```
int  raise(int sig);
```

This function raises signal **sig** and returns 0.

rex.

```
void rex(char cmd);
```

This function returns control to REX. If *cmd* is "h", the system halts. If *cmd* is "b", a boot command with no arguments is performed. If *cmd* is "r", a system dependent restart operation is requested.

setenv.

```
int setenv(char *name, char *value);
```

This function sets the environment variable name to **value**. It returns 0 if successful.

setjmp.

```
typedef int jmp_buf[12];
int setjmp(jmp_buf env);
```

This function stores the current context in **env** and returns 0. The subsequent execution of *longjmp* with **env** as an argument causes a second return from this function with a nonzero value.

showfault.

```
void showfault(void);
```

This function displays system-specific hardware fault data from the most recent exception on the console.

signal.

```
typedef void (*sig_handler)(int);
  sig_handler *signal(int sig, sig_handler func);
```

This function specifies the address of the handler routine for signal **sig** and returns the address of the previous handler if successful, otherwise it returns **SIG_ERR**(-1). If **func** is **SIG_DEF**(0) then the default handler is specified. If **func** is **SIG_IGN**(1) then the signal is ignored. Otherwise, **func** must specify the address of a handler routine that returns **void** and which takes a single **int** argument. REX calls this routine when signal **sig** is raised, passing **sig** as the argument to the routine.

When a signal is raised, the handler reverts to the default handler. The default handler reinitializes REX as if a system reset occurred.

REX raises three signals: **SIGINT**(4) when Ctrl-c is typed, **SIGSEGV**(5) when a segmentation exception is detected, and **SIGBUS**(7) when a bus error is detected.

slot_address.

```
unsigned long *slot_address(int sn);
```

This function returns the base address of a TURBOchannel option in slot **sn**.

sprintf.

```
int sprintf(char *s, char *format, ...);
```

This function is identical to *printf* except that the output is written to the string **s** instead of the console.

strcat.

```
char *strcat(char *s1, char *s2);
```

This function appends string **s2** to the end of string **s1**. It returns **s1**.

strcmp.

```
int strcmp(char *s1, char *s2);
```

This function compares successive characters of two strings, **s1** and **s2**, until it finds characters that are not equal. If all characters are equal, it returns zero. If the different character from **s1** is greater than that from **s2** (both taken as **unsigned char**), this function returns a positive number, otherwise it returns a negative number.

strcpy.

```
char *strcpy(char *s1, char *s2);
```

This function copies the string **s2** to the string **s1**. It returns **s1**.

strlen.

```
int strlen(char *s1);
```

This function returns the number of characters in string **s1** *not* including its terminating null character.

strncat.

```
char *strncat(char *s1, char *s2, int n);
```

This function appends the string **s2**, *not* including its terminating null character, to the string **s1**. The function copies no more than **n** characters from **s2**. It then stores a null character as the last element of **s1**. It returns **s1**.

strncmp.

```
int strncmp(char *s1, char *s2, int n);
```

This function compares successive characters of two strings, **s1** and **s2**, up to character **n**. If all compared characters are equal, it returns zero. If the different character from **s1** is greater than that from **s2** (both taken as **unsigned char**), this function returns a positive number, otherwise it returns a negative number.

strncpy.

```
char *strncpy(char *s1, char *s2, int n);
```

This function copies the string **s2**, *not* including its terminating null character, to the string **s1**. It copies no more than **n** characters from **s2**. The function then stores a null character as the last element of **s1**. It returns **s1**.

strtol.

```
long strtol(char *s, char **endptr, int base);
```

This function converts the initial characters of the string **s** to a value of type **long**. If **endptr** is not of value **NULL**, the pointer to the unconverted portion of **s** is stored in ***endptr**. If **base** is zero, the function determines the base from the contents of the string. A leading 0x or 0X indicates base 16, a leading 0 indicates base 8, otherwise the base is 10. If the string has an improper format, **s** is stored in ***endptr** and the value returned is zero.

testintr.

```
int testintr(int sn);
```

This function checks for an interrupt pending from a TURBOchannel option module in slot **sn**. If an interrupt is pending, 1 is returned, otherwise 0 is returned. REX runs with all interrupts masked so this function only indicates that an interrupt is pending, REX does not provide an interrupt "delivery" mechanism.

The *enableintr* routine must be called to enable interrupts from an option module before testing for an interrupt. The *disableintr* routine must be called to disable interrupts when the testing is completed.

time.

```
long time(long *tod);
```

If **tod** does not equal **NULL**, this function stores the time relative to a system-specific base time in ***tod**. This function returns the relative time value. The least significant bit of the time value corresponds to one second.

unsetenv.

```
int unsetenv(char *name);
```

If the environment variable **name** is defined, this function removes it from the environment table and returns 0, otherwise it returns a nonzero value.

wbflush.

```
void wbflush(void);
```

This routine waits for the system write buffer (if any) to be written out before returning.

Callback Vector. The callback vector is a vector of pointers to routines. Table 3 lists the contents of the callback vector.

Table 3. Callback Vector Contents

Offset	typedef	Description
54	int	(*bootinit)( );
58	int	(*bootread)( );
5c	int	(*bootwrite)( );
7c	void	(*clear_cache)( );
98	int	(*console_init)( );
88	int	(*disableintr)( );
8c	int	(*enableintr)( );
a8	int	(*execute_cmd)( );
84	int	(*getbitmap)( );
24	int	(*getchar)( );
64	char	*(*getenv)( );
28	char	*(*gets)( );
80	int	(*getsysid)( );
a4	tcinfo	*(*gettcinfo)( );
9c	void	(*halt)( );
38	int	(*io_poll)( );
78	void	(*leds)( );
50	void	(*longjmp)( );
00	void	*(*memcpy)( );
04	void	*(*memset)( );
74	void	(*msdelay)( );
2c	int	(*puts)( );
30	int	(*printf)( );
44	int	(*raise)( );
ac	void	(*rex);
60	int	(*setenv)( );
4c	int	(*setjmp)( );
a0	void	(*showfault)( );
40	sig_handler	(*signal)( );
6c	unsigned long	(*slot_address)( );
34	int	(*sprintf)( );
08	char	*(*strcat)( );
0c	int	(*strcmp)( );
10	char	*(*strcpy)( );
14	int	(*strlen)( );
18	char	*(*strncat)( );
20	int	(*strncmp)( );
1c	char	*(*strncpy)( );
3c	long	(*strtol)( );
90	int	(*testintr)( );
48	long	(*time)( );
68	int	(*unsetenv)( );
70	void	(*wbflush)( );
94	void	*private; pointer to system-specific data
		b0 to d4 reserved