EK-TCAAD-FS-004

TURBOchannel

Firmware Specification

On-line version.

Previous versions of this document are obsolete and should be discarded. This document supersedes all previous versions.

January 1993

TURBOchannel Specifications are controlled by the TURBOchannel Industry Group. For information about how to order Specifications call

Digital Equipment Corporation

Telephone: 508.493.7044
FAX: 508.493.3692

Examples provided in this specification are based on the DECstation/DECsystem 5000 Model 200 but can apply to other TURBOchannel systems.

TURBOchannel Technology Transfer Agreement

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 Specifications*) 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 in 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.

Important Changes to the TURBOchannel Firmware Specification

This version of the *TURBOchannel Firmware Specification* includes all changes made since the original specification was released. Revision bars mark technical changes added since Version 2C of the specification.

Changes in Version 3

- **Block I/O Write**: Add Block I/O Write callbacks to enable TURBOchannel Firmware to work on a system that supports Block I/O Writes in hardware.
- **Parity**: Add parity callbacks to enable TURBOchannel options to test and control parity logic.
- **malloc( ) Callback**: Add a malloc( ) callback for options requiring more than 64Kb of temporary buffer space.
- **System Firmware Revisions**: Add a System Firmware Revision section to describe the different system firmware capabilities.
- **REX Memory**: Add an extra REX memory region (region 4) to standardize the use of extra memory.
- **getsysid( ) Callback**: Change the getsysid( ) callback to determine what capabilities are available in System Firmware.
- **Firmware**: Revision section added with coding examples.

Changes in Version 2C

- **Option Module Firmware**: The introduction of this chapter now contains an overview of option module address space.
- **ROM Objects**: This section has been corrected to show that the ROM objects section is required.
- **msdelay**: This section has been expanded.

Changes in Version 2B

- **Important Changes to the TURBOchannel Firmware Specification**: This section has been added.
- **Standard REX Commands**, section "**cnfg** Command", specifies the DECstation/DECsystem 5000 Model 200 as the source of the **cnfg** display.

Conventions Used in This Specification

- **Characters in boldface type** represent REX commands, ROM objects, and system module scripts and routines.
- ***Characters in boldface italic type*** represent named variables.
- *Terms in italic type* represent TURBOchannel signals and variables that are replaced with actual values.
- Quotation marks (" ") indicate quoted strings.
- `Monospace type` is used for program call interfaces and to show text displayed on a monitor.
- The number sign (#) represents the number of a slot. Firmware uses these numbers to identify the location of a TURBOchannel module.
- Square brackets ([ ]) surround optional arguments.
- Ellipses (...) follow an argument that can be repeated.

Call Prototype

This document defines all program call interfaces in ANSI C. The example defines the call interface to the **strncat** routine. This interface is called a call prototype.

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

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.

Hexadecimal Offsets

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

Contents

4 System Module Standard Scripts

5 System Module Callback Routines

6 Firmware Revision Descriptions

Figures

Tables

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 components are specified in the *TURBOchannel Hardware Specification*.

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. This channel is asymmetrical:

- The system module has read or write access to an option module.
- An option module has read or write access to the system module.
- An option module has no access to other option modules.

Figure 1-1 shows the access permitted between a TURBOchannel system module and TURBOchannel option modules.

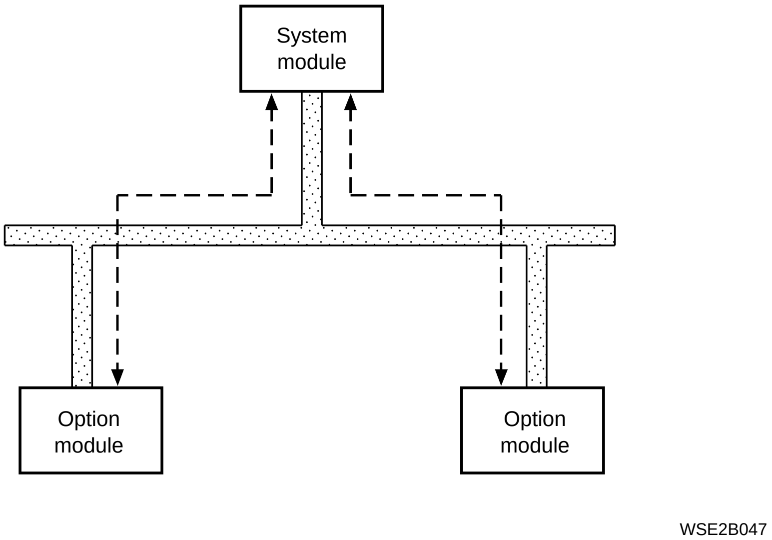

Figure 1-1. TURBOchannel I/O access

TURBOchannel Slots

TURBOchannel systems divide a part of the system physical address space into slots. 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 1-2 shows how physical address space in a TURBOchannel system is divided into slots.

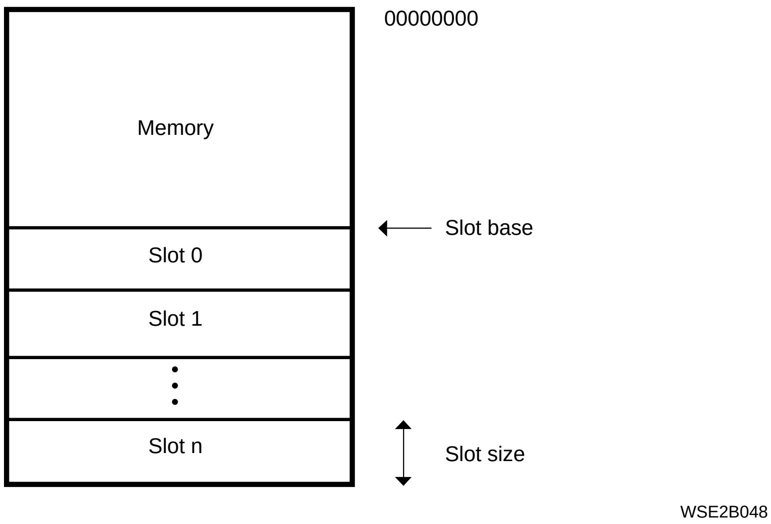

Figure 1-2. TURBOchannel slot address space

The slot number of the system module is implementation-specific, but typically the system module is assigned the highest slot number. The slot number assigned to an option module depends on the system module connector used by the option. Some systems may implement fewer option module connectors than there are slots. The missing slots can be used for integral options (components that appear to be separate TURBOchannel options but are part of the system module) or can remain unused.

TURBOchannel Module ROMs

The system module ROM contains system module firmware. An option module ROM contains information about the option module, including the ROM geometry. The ROM 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 diagnostic and initialization routines
- To boot the system

- To examine and modify memory
- To perform other system management functions

REX commands can be entered at the system console or read from scripts stored in ROM.

ROM Objects

REX makes extensive use of ROM objects. ROM objects contain module-specific scripts and firmware that are stored in ROM. REX uses ROM objects to perform module-dependent operations such as bootstraps and diagnostic routines. For example, to boot a program from a disk controller module, REX loads the ROM object named **boot** into memory from the controller module and then calls **boot** to load the program from the disk.

REX performs all module-specific functions using firmware or scripts from the modules. This allows REX to remain device independent, and allows new modules to be easily developed and added to TURBOchannel systems.

Diagnostics

REX invokes all system and option module diagnostic routines, except for system module diagnostic routines that test system resources required by REX. (These diagnostic routines are executed directly from the system module ROM before REX is invoked.) Diagnostic routines are module-specific and are stored in a test object. REX loads the test object into memory and calls the object to perform module diagnostic routines.

Diagnostic routines can be divided into smaller routines that are used to test parts of a module. When a test fails, the module number and the test routine name are displayed, allowing the failing unit to be identified.

REX uses standard scripts from each module to invoke diagnostic routines. When REX is invoked after system power-up, REX automatically calls the system module **powerup** script. The **powerup** script calls the power-up self-test script **pst-t** or **pst-q**, which run power-up tests of the system module and each option module. **pst-t** performs a thorough test; **pst-q** performs a quick test.

Additional diagnostic scripts that perform special tests not used at power-up can be included in module ROMs and run manually.

Bootstraps

REX invokes option bootstrap programs. A bootstrap program loads a program into memory from a boot module. A disk controller or an Ethernet controller are typical modules that could be used as boot devices. To perform a bootstrap, REX loads the **boot** object from the boot module ROM and then calls that object to load the program. After the program is loaded, REX transfers control to the program.

System Console

REX uses the system console to display status messages and accept operator commands. REX configures a device or pair of devices as the system console. Modules that act as console devices contain console drivers that are used by REX to communicate with the device. Console device assignments are made by the operator or by a system-specific autoconfigure procedure. These assignments are stored in the system module's nonvolatile memory.

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 relate to the interface between system module firmware and option module firmware (see Chapter 3).

Control is passed to the system module firmware when a hardware reset occurs and when software explicitly branches to system module firmware. (Branching is treated in the same way as a hardware reset.) A hardware reset occurs at power-up; a hardware reset can also occur in response to a major system error or operator action.

The ROM Executive (REX) program receives control on reset. When invoked, REX checks the cause of the reset.

- If the reset was caused by power-up, REX performs system module power-up testing and then calls all module-specific power-up tests. When power-up testing is complete, REX initializes the system and option modules.
- If the reset is not due to power-up, REX performs only the initialization.

REX implements a small set of standard commands that can be entered either at the system console or from a script. REX performs option module diagnostics by reading standard diagnostic scripts that contain commands that run module-specific diagnostics. REX is also responsible for configuring the system console.

Standard REX Commands

Standard REX commands are responsible for

- Bootstrapping the system
- Displaying system configuration information
- Initializing the system
- Performing system diagnostics

These commands cause TURBOchannel module-specific firmware to be executed. REX commands are accepted either from the system console or from scripts.

The following sections describe the standard REX commands: **boot**, **cnfg**, **init**, and **t**. When entering standard REX commands note the following:

- Type the command at the prompt `(>>)`.
- *Letters in italic type* are variables that are replaced with actual values.

- Items enclosed in square brackets ([ ]) are optional.
- Ellipses (...) follow an argument that can be repeated.
- The number sign (#) represents the slot number of a module.
- Press Return to enter the command.

boot Command

>>**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*. *#* is the slot number of the module acting as the boot device and *path* is a device-specific file specification. The **-n** option suppresses program execution after the program has been 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. A system-specific command can be used to set the value of the ***boot*** environment variable. The interpretation of *argument* is module-specific.

For example, to load and execute the file vmunix on a disk drive with SCSI ID 0 that is part of the SCSI bus connected to option slot 5, type `boot 5/rz0/vmunix`.

cnfg Command

>>**cnfg** [*#*]

The **cnfg** command displays system configuration information. If a slot number (*#*) is specified, detailed configuration information for the module connected to that slot is displayed. If no slot number is specified, brief configuration information for each module in the system is displayed.

This example shows a **cnfg** display with no slot number specified. The following display shows a DECstation/DECsystem 5000 Model 200 with optional Ethernet, SCSI, and color frame buffer modules.

```
>>cnfg
 7:  KN02-AA  DEC  V5.3c  TCF0  ( 24 MB)
 6:  PMAD-AA  DEC  V5.3a  TCF0  (enet: 08-00-2b-0c-e0-d1)
 5:  PMAZ-AA  DEC  V5.3b  TCF0  (scsi = 7)
 2:  PMAD-AA  DEC  V5.3a  TCF0  (enet: 08-00-2b-0f-43-31)
 1:  PMAZ-AA  DEC  V5.3b  TCF0  (scsi = 7)
 0:  PMAG-BA  DEC  V5.3a  TCF0  (CX -- d=8)
```

The following example shows a **cnfg** display with slot number 7 specified. This display shows the system and memory modules in slot number 7 of a DECstation/DECsystem 5000 Model 200:

```
>>cnfg 7
 7: KN02-AA  DEC      V5.3c    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 are module-specific. The following example shows a **cnfg** display for a SCSI controller module in slot 5 of a DECstation/DECsystem 5000 Model 200.

```
>>cnfg 5
 5: PMAZ-AA  DEC     V5.3b    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 Command

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

The **init** command initializes module hardware. If module number (#) is specified, only that module is initialized. If no module number is specified, all modules are initialized. The interpretation of *argument* is module-specific.

t Command

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

OPTIONS	**-l**	loop

The **t** command runs module tests. A test is specified by *#/testname*. *#* 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. The interpretation of *argument* is module-specific.

System-Specific REX Commands

Additional system-specific REX commands are both allowed and expected. For example, 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 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 retained in nonvolatile RAM; others are lost when REX exits.

REX performs environment variable substitution when reading commands from scripts. REX uses double quotation marks (") and single quotation marks (') as quote delimiters. If the script contains the string $X or ${X} between double quotation marks, REX replaces the string with the contents of the environment variable ***X***. If either string appears between single quotation marks, substitution is not performed.

Table 2-1 shows the standard environment variables. Additional environment variables can be set as implementation-specific side effects of various bootstrap and test procedures.

Table 2-1. Standard Environment Variables

Variable	Function
boot[1]	Specifies the default arguments for the **boot** command.
console[1]	Controls the choice of the system console. any value other than "s" - the system autoconfigures the console[2]. "s" - the system uses a terminal connected to a system module. Setting ***console*** causes the system to immediately reconfigure and initialize the system console.
haltaction[1]	Specifies system actions after halt. "b" - the system performs a **boot** command after halt. "h" - causes the system to halt (the system can accept commands from the system console). "r" - the system performs a restart. If the restart fails, the system performs a **boot** command.
more[1]	Contains the screen height in lines. If this value is nonzero, the system paginates all command output using the value as the page size.
osconsole	Contains the slot numbers of the console drivers. If a tty driver from slot x is used as the system console, ***osconsole*** is set to "x". If a CRT driver from slot y and a keyboard driver from slot z are used as the system console, ***osconsole*** is set to "y,z".
testaction[1]	Specifies the type of power-up self-test that the system runs. "t" - specifies a thorough test of the system. "q" - specifies a quick test of the system. "m" - specifies that manufacturing-specific tests are performed[3].
#	Specifies the slot number of the module that contains the current script. If no script is active, the base system module is specified.

[1]Environment variables preserved in nonvolatile RAM.

[2]The autoconfigure process is system-specific.

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

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 five regions: 0, 1, 2, 3, and 4. Table 2-2 lists the addresses and use of each memory region in REX.

Table 2-2. REX Memory Regions

Region	Starting Address	Ending Address	Use
0	0xa0000000	0xa000ffff	Restart block, exception vectors, REX stack and bss
1	0xa0010000	0xa0017fff	Keyboard or tty drivers
2	0xa0018000	0xa001f3ff [1]	CRT driver
3	0xa0020000	0xa002ffff	**boot**, **cnfg**, **init**, and **t** objects
4	0xa0030000	0xa003ffff	64KB scratch space

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

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

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

Region 4 is volatile in the sense that the option firmware must not rely on these locations to remain valid between testing of TURBOchannel devices. It is the option firmware's responsibility to initialize region 4 before using it, if needed. System Firmware will not explicitly clear this region before loading any ROM object, nor will system firmware directly load this region from a TURBOchannel ROM. Region 4 must not be used by region 1 or region 2 resident code (console drivers).

Program Interface

When REX transfers control to a program in memory because 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)
```

When ***argc*** and ***argv*** have 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 that are used for

- Booting
- Displaying configuration information
- Performing console input and output
- Performing module-specific initialization
- Performing module-specific tests

All option module firmware is stored in objects in the option module ROM. Scripts are also stored in objects in the option module ROM. Scripts are used to organize diagnostic test sequences.

Option module address space is divided into four sections:

- 0x000 to 0x3DF - Option address space. This space is available to option designers.
- 0x3E0 to 0x47F - ROM header.
- 0x480 to an option-specific size - ROM objects section.
- The end of the ROM objects section to the end of the slot - Option address space. This space is available to option designers and is dependent on the slot size shown in the *TURBOchannel System Parameters*.

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 0x3E0; 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. The ROM header contains information used by the system module firmware and the operating system to determine

- Whether a module is present
- What type of module is present

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

- Critical ROM geometry information

The ROM header should not be accessed for the first 200 milliseconds following system reset [2]. Figure 3-1 and Table 3-1 show the assignment of fields in the ROM header. Note that only the first byte of each word of the ROM header contains information. The first 0x3E0 (992 decimal) bytes of the board address space is available for use by the designer.

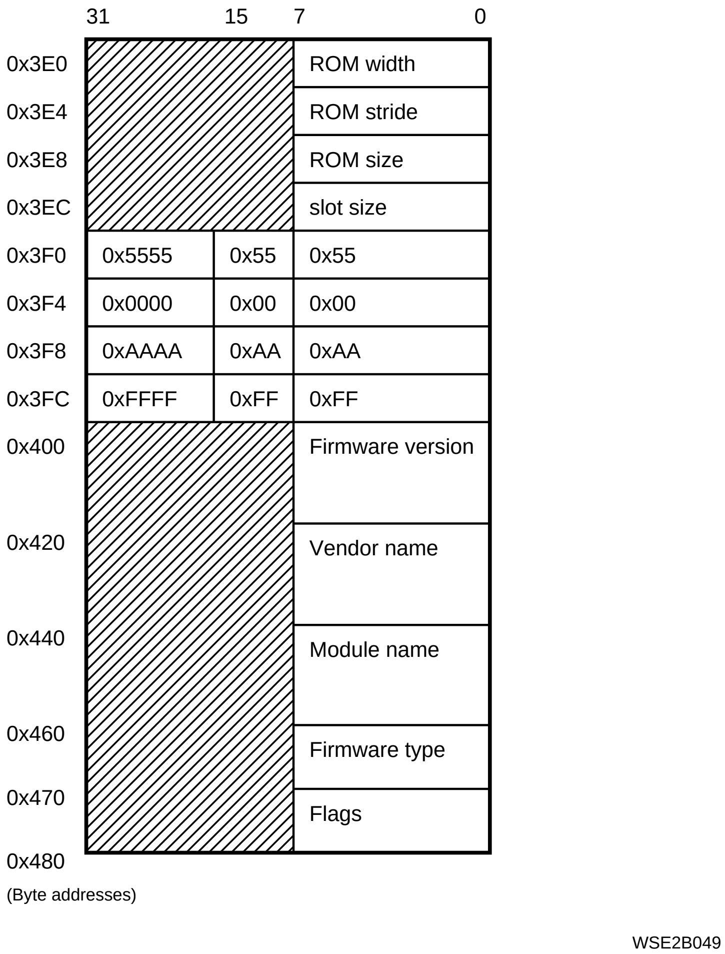

Figure 3-1. ROM header locations

[2] For some older TURBOchannel options, the ROM header does not appear valid during this period.

Table 3-1. ROM Header Fields

Field	Contents
ROM width	ROM geometry information field; **ROM width** is the width of the ROM in bytes. This value can be 1, 2, or 4.
ROM stride	ROM geometry information field; **ROM stride** is the address stride of the ROM as seen by the system module. This value must be 4.
ROM size	ROM geometry information field; **ROM size** is the size of the ROM divided by 8192.
Slot size	Contains the minimum TURBOchannel slot size required by the option module divided by 4,194,304 (4 megabytes). For example, the DECstation/DECsystem 5000 Model 200 **slot size** is 1.
55, **00**, **AA**, **FF**	The four pattern fields are used for test purposes and as part of the signature of a valid ROM header.
Module firmware version	Contains 8 ASCII characters of version information. Only graphic ASCII codes (that is, values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.
Module vendor name	Contains 8 ASCII characters indicating the module manufacturer. Only graphic ASCII codes (that is, values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.
Module name	Contains 8 ASCII characters indicating the module name. Only graphic ASCII codes (that is, values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.
Firmware type	Contains 4 ASCII characters indicating the type of firmware present in the module ROM. Only graphic ASCII codes (that is, values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks. Option modules which are to remain compatible with TCF0 System Firmware must contain "TCF0" (TCFzero) in this field. See Section 6 for a description of System Firmware Revisions. See Section 6 for a description of System Firmware Capabilities and how to maintain backward compatibility.
Flags	A 4-byte field with bit [0] indicating whether the module implements parity. When the bit is 1, the option module implements TURBOchannel parity. When the bit is 0, the option module does not implement TURBOchannel parity. All remaining bits are reserved and must be 0.

ROM Objects

A ROM object contains header information and an object body. REX uses the header information to locate objects, determine the object type, and determine where to load firmware objects into RAM. The object body contains either code or text. Figure 3-2 and Table 3-2 show field assignments in the ROM object section. The minimum requirement for the ROM objects section is a single object with a length of 0. This object indicates the end of the ROM objects section.

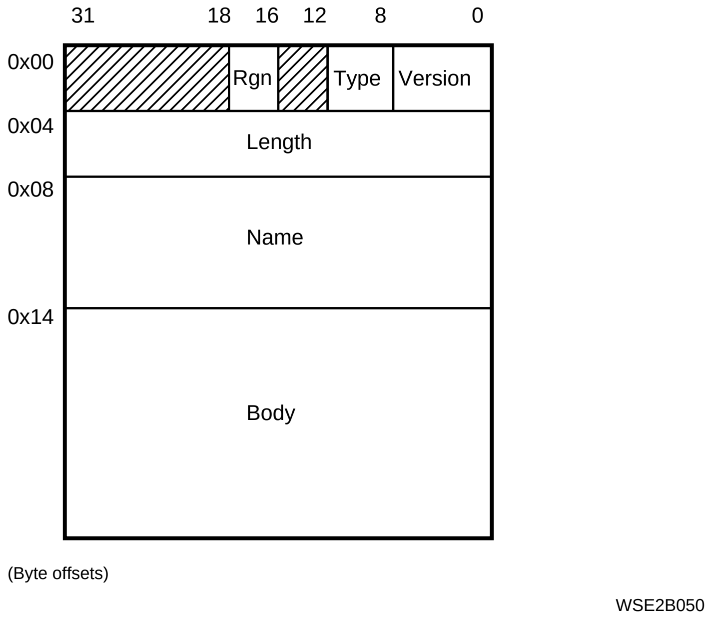

Figure 3-2. ROM object fields

Table 3-2. ROM Object Fields

Field	Contents
Rgn	A 2-bit field that specifies the region into which a code object is loaded for execution. Valid values are 1, 2, and 3. Refer to the "REX Memory Regions" section for more information.
Type	A 4-bit field containing a code that indicates the object body type. Valid **type** values are 0, 4, and 5. All other values for **type** are reserved. 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 an ASCII string terminated by 0 and stored in little-endian byte order. 5 - Symbolic Link. The body of this type of object contains an ASCII string terminated by 0 and stored in little-endian byte order. The string contains the name of another object. Symbolic links can be used to redirect a reference from one object name to another. This redirection is allowed only between objects in the same module; redirection to another module is not allowed.
Version	An 8-bit field containing the ROM object version number.
Length	A 32-bit field containing the length in bytes of the object, including the header information. The length must be a multiple of 4. A special object with a length of zero terminates the ROM objects section of the ROM.
Name	A 12-byte field that contains the name of the object in ASCII format. Only graphic ASCII codes (that is, values from 0x20 to 0x7e) are allowed. Unused characters must contain blanks.
Body	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 between objects are not allowed. A special object with a length of zero terminates the ROM objects section of the ROM. If no other ROM objects are provided this object is the only entry.

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; the firmware is always copied to RAM for execution.

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

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

When ***argc*** and ***argv*** have standard meanings, ***slot*** is the slot number of the module and ***vector*** is a pointer to the callback vector [1].

Table 3-3 lists the seven standard ROM executable objects.

Table 3-3. ROM Executable Objects

ROM Object	Used By
kbd, **tty**, **crt**,	The console terminal interface; the **initc**, **getc**, and **putc** functions
boot	The REX **boot** command
cnfg	The REX **cnfg** command
init	The REX **init** command
t	The REX **t** command

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

When REX configures the console terminal interface, REX loads the appropriate console driver object, **kbd**, **tty**, or **crt**, into memory for use. REX calls the **initc** function of the object once before calling the **getc** or **putc** function. 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 executing a **boot** command, REX first loads the **boot** object, then calls the **boot** function to perform the boot. REX follows the same procedure when executing a **cnfg** command, an **init** command, and a **t** command. Note that 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.

The following sections describe standard option module routines.

[1] For more information refer to *The C Programming Language*, Kernighan and Ritchie, Prentice Hall, 1988

boot Function

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

This function loads a program into RAM. **boot** returns the execution address of the program if the load succeeds. If the load is not successful, **boot** returns 0.

The REX **boot** command loads the **boot** object from the specified module ROM and calls **boot** 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 **bootinit**, **bootread**, and **bootwrite** routines. The **boot** object must store the addresses of these routines in the callback vector. When REX calls the program, REX passes the address of the callback vector to the boot object. Placing the addresses of the **bootinit**, **bootread**, and **bootwrite** routines in the callback vector makes them available to the programs in the bootstrap sequence. The specifications of these routines are in Chapter 5.

Optionally, 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 [1].

cnfg Function

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

This function displays configuration data about a module. **cnfg** returns 0 if successful. If not successful, **cnfg** returns a negative value.

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

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

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

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

[1] This argument has been used by existing software as part of a crash-dump mechanism. The crash-dump code issues a boot request with **-Noboot** specified to load the boot routine and then uses the **bootinit**, **bootread**, and **bootwrite** routines to dump the contents of memory.

```
>>cnfg
 7:  KN02-AA  DEC  V5.3c  TCF0  ( 24 MB)
 6:  PMAD-AA  DEC  V5.3a  TCF0  (enet: 08-00-2b-0c-e0-d1)
 5:  PMAZ-AA  DEC  V5.3b  TCF0  (scsi = 7)
 2:  PMAD-AA  DEC  V5.3a  TCF0  (enet: 08-00-2b-0f-43-31)
 1:  PMAZ-AA  DEC  V5.3b  TCF0  (scsi = 7)
 0:  PMAG-BA  DEC  V5.3a  TCF0  (CX -- d=8)
```

If ***argc*** is 2, the **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     V5.3c    TCF0   ( 24 MB)
        mem( 0): a0000000:a07fffff   (  8 MB)
        mem( 1): a0800000:a0ffffff   (  8 MB)
        mem( 2): a1000000:a17fffff   (  8 MB)
```

getc Function

```
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, the character 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 Function

```
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 **init** to perform module-specific initialization. If no **init** object is found in the option module ROM, no error occurs; 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 successful. If not successful, **init** returns a negative value.

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

For options which support parity, the *init* function must perform the following sequence: call getsysid( ), if byte 1 (bits [15..8]) from the 32-bit word returned indicates the system firmware supports TCF1, then the option firmware may enable parity detection on the system side and then on itself. Option firmware enables parity on the system side by calling the enable_parity( ) callback. Option firmware enables parity on itself by writing to an option-specific CSR. Other ROM objects may *optionally* enable parity, but if parity is to be enabled, it *must* at least be done in the *init* function. See Chapter 6 for code examples.

For options which support Block I/O Writes, the *init* function must perform the following sequence: call getsysid( ), if byte 1 (bits [15..8]) of the 32-bit word returned indicates the System Firmware supports TCF1, then the Option Firmware must call enable_block_mode( ) to request an option-specific Block I/O Write transfer size.

initc Function

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

This function initializes the console driver. If initialization is successful, **initc** returns 0. If initialization is not successful, **initc** returns a negative value.

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

putc Function

```
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, **putc** outputs the character and 0 is returned. If output is not possible, a negative value is returned.

t Function

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

The REX **t** command loads the object named **t** into memory and calls **t** to invoke a module-specific test. If the **t** test object is not found, the **t** command fails. **t** returns 0 if successful. If not successful, **t** returns a negative value:

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

The value of ***argc*** can be 2 or greater. The value of ***argv[0]*** is "t". The value of ***argv[1]*** is a string with the format *#/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**. If any of these scripts are not present in a module, the corresponding power-up testing of the module is not performed. REX invokes these scripts automatically at system power-up. Additional scripts can be present and run from the console terminal. Option module scripts must contain only REX **t** commands. System module scripts can contain any standard REX command as well as any system-specific command. All of these scripts are optional.

pst-t Script

This script is executed on power-up when the ***testaction*** environment variable is set to "t". **pst-t** invokes thorough module diagnostics. In the example, **pst-t** invokes 10 diagnostic tests. Note that the TURBOchannel slot number is specified by `${#}`. During processing, REX replaces `${#}`with the value of the **#** environment variable, which always contains the TURBOchannel slot number of the script currently executing. This replacement is required because a module can be connected to any slot.

```
t ${#}/regs
t ${#}/ram
t ${#}/esar
t ${#}/int-1b
t ${#}/ext-1b
t ${#}/crc
t ${#}/cllsn
t ${#}/promisc
t ${#}/m-cst
t ${#}/int
t ${#}/reg
```

pst-q Script

This script is executed on power-up when the **testaction** environment variable is set to "q". **pst-q** also invokes module diagnostics. **pst-q** is similar to the **pst-t** script, but **pst-q** should be faster. Either fewer tests are executed, faster tests are executed, or both. In the following example, note that some tests shown in the **pst-t** example have been removed, presumably tests that take a long time to execute. A reasonable time for a **pst-q** script to complete execution is less than six seconds.

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

cnsltest Script

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

5

System Module Callback Routines

The system module implements a set of routines that are available to option module ROM firmware via a callback vector. The address of the callback vector is passed to all ROM object firmware and to programs loaded by the **boot** command. The routines are provided for general utility or for isolating 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 following sections describe callback routines. Table 5-2 shows callback vector contents.

bootinit Routine

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

This routine is called to initialize the bootstrap input/output routines **bootread** and **bootwrite**. **bootinit** must be called before using **bootread** or **bootwrite**. This routine returns 0 if successful. If not successful, **bootinit** returns a negative value. (See the section "Option Module Firmware" in Chapter 3 for more information about this routine.)

bootread Routine

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

This function uses the boot driver to read ***n*** bytes 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). If the boot device is not block structured, the interpretation of ***b*** is device-specific and is typically ignored. The routine returns the number of bytes read if successful. If an error is detected **bootread** returns a negative value. (See the section "Option Module Firmware" in Chapter 3 for more information about this routine.)

bootwrite Routine

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

This function uses the boot driver to write ***n*** bytes 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). If the boot device is not block structured, the interpretation of ***b*** is device-specific. The routine returns the number of bytes read if successful. If an error is detected **bootread** returns a negative value. (See the section "Option Module Firmware" in Chapter 3 for more information about this routine.)

clear_cache Routine

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

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

console_init Routine

```
int console_init(void)
```

This routine initializes the console I/O routines. **console_init** returns 0 if successful. If not successful, **console_init** returns a negative value.

disableintr Routine

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

This function disables interrupts from the TURBOchannel option module in slot ***sn***. **disableintr** returns 0 if successful. If not successful, **disableintr** returns a negative value. See the section "**testintr** Routine" for additional information.

disable_parity Routine (TCF1)

```
int disable_parity(int sn);
```

This function disables parity checking by the System TURBOchannel interface for slot **sn**. The routine **disable_parity** returns 0 if successful, a negative number if the system does not support TURBOchannel parity.

enableintr Routine

```
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 whether an interrupt is pending. This function returns 0 if successful. If not successful, **disableintr** returns a negative value. See the section "**testintr** Routine" for additional information.

enable_block_mode Routine (TCF1)

```
int enable_block_mode(int sn, int n_words);
```

This function enables block mode I/O write transfers on the System TURBOchannel interface for slot **sn**, if the requested Block Mode I/O Write transfer size is compatible with the Option-requested maximum transfer size **n_words**. If System Firmware determines the value of **n_words** to be compatible with System Hardware, then **enable_block_mode** returns N, the maximum Block I/O Write transfer size (of 32-bit words) it has agreed to. Note that the transfer size agreed to by the system may be less than the requested transfer size. Otherwise, **zero** is returned to indicate that the System Hardware does not support a compatible Block I/O Write transfer size with the size requested by Option Firmware. Block mode may be disabled by passing the **n_words** parameter equal to zero. If an option supports Block I/O Write capability, then it must enable it from its *init* function.

Note: This functionality is new to TCF1 System Firmware, see Chapter 6 for a description of TCF1 System Firmware Capabilities.

enable_parity Routine (TCF1)

```
int enable_parity(int sn);
```

This function enables parity checking by the System TURBOchannel interface for slot **sn**. The routine **enable_parity** returns 0 if successful, a negative number if the system does not support TURBOchannel parity. If option software is to enable parity it must do so, as a minimum, when its *init* function is called.

execute_cmd Routine

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

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

getbitmap Routine

```
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***. **getbitmap** returns the number of bytes in ***bitmap***. If ***map*** is **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 that specifies memory availability. If bit_i of $byte_j$ is 1, $page_{8j+i}$ is available. If this bit 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 Routine

```
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, **getchar** returns **EOF**. Input characters are not echoed.

getenv Routine

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

If the environment variable ***name*** is defined, this function returns a pointer to the value of ***name***. If ***name*** is not defined, **getenv** returns **NULL**. The pointer points to a temporary string which may be destroyed by subsequent calls to **setenv** or **getenv**.

gets Routine

```
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**, **gets** returns **s**. If no characters are stored *gets* returns **NULL**. No means of limiting the number of characters that this routine reads and stores are provided. Characters are echoed as they are read and system-specific line editing functions can be performed.

getsysid Routine

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

This function returns a value containing the System Firmware revision level. Bits [15..8] contain the System Firmware Revision level:

```
Bits [15..8]          System Firmware level
    0x02                    TCF0
    0x03                    TCF1
```

All of the remaining bits, that is, bits [31..16] and [7..0], should be ignored by Option Firmware. See Section 6.2 for a description of TCF1 System Firmware capabilities.

gettcinfo Routine

```
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 5-1 lists the TURBOchannel parameters in this structure and gives sample parameters from a DECstation/DECsystem 5000 Model 200. Remaining fields are reserved for future use.

Table 5-1. gettcinfo Implementation Parameters

Parameter	Description	DECstation/DECsystem 5000 Model 200
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

(continued on next page)

Table 5-1 (Cont.). gettcinfo Implementation Parameters

Parameter	Description	DECstation/DECsystem 5000 Model 200
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

halt Routine

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

This function is called to halt the system. When the ***cnt*** values whose first address is ***v*** are printed on the console, 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 Routine

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

This function is called to allow REX to check for pending console input or output. **io_poll** raises **SIGINT (4)** if Ctrl-c is entered. In addition, **io_poll** enables all console output if Ctrl-q is entered, and disables all console output if Ctrl-s is entered. This function returns a nonzero value if any console input is pending. If no console input is pending, **io_poll** returns 0.

leds Routine

```
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 0. **leds** should be used by option modules for debugging purposes, not for error reporting.

longjmp Routine

```
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 0, **setjmp** returns ***value***. If ***value*** is 0, **setjmp** returns 1.

malloc Routine (TCF1)

```
void *malloc(int n_bytes);
```

This function allocates an **n_byte** buffer and returns a pointer to the start of the buffer. The function **malloc** returns the start address of the buffer if the buffer was allocated, otherwise it returns 0. The 32-bit buffer address is such that the System physical address can be obtained from the address returned by **malloc** by clearing bits [31..29] of the address returned. Memory is automatically freed (but not overwritten) when control returns to the System Firmware. Option firmware *console drivers* and *boot functions* **must not** call malloc. This callback is available in TCF1 System Firmware, see Chapter 6 for a description of TCF1 System Firmware Capabilities. The maximum buffer size that may be allocated is system dependent.

memcpy Routine

```
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***. **memcpy** returns ***s1***.

memset Routine

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

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

msdelay Routine

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

<This routine waits for ***delay*** milliseconds to elapse before returning. The accuracy of **msdelay** is system dependent. The cumulative error may become significant for large values of *int delay*. Therefore, a developer should not attempt to use the **msdelay** routine as a real-time clock.

TURBOchannel option functions being timed by the use of the **msdelay** routine may vary in duration when run on different TURBOchannel based systems. Duration differences may be observed when the time to complete that module's function is dependent upon any or all of the following:

- TURBOchannel bus speed
- DMA throughput
- CPU speed
- Memory access speed
- System architecture.

Consequently, option developers implementing firmware using the **msdelay** routine tuned to a specific system platform may find that ***delay*** (the elapsed time before returning) is insufficient for the option to complete its task if it was installed in another TURBOchannel-based platform. <

puts Routine

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

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

printf Routine

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

This function generates formatted output to the console. **printf** returns the number of characters generated; if an output error occurs **printf** returns a negative value. 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 Routine

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

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

rex Routine

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

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

setenv Routine

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

This function sets the environment variable ***name*** to ***value***. **setenv** returns 0 if successful. If not successful, **setenv** returns a nonzero value.

setjmp Routine

```
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 **setjmp** with a nonzero value.

showfault Routine

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

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

signal Routine

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

This function specifies the address of the handler routine for the *sig* signal and returns the address of the previous handler if successful. If not successful, **signal** returns **SIG_ERR**(-1). If ***func*** is **SIG_DEF**(0) the default handler is specified. If ***func*** is **SIG_IGN**(1) the signal is ignored. Otherwise, ***func*** must specify the address of a handler routine that returns ***void*** and that takes a single ***int*** argument. REX calls this routine when the *sig* signal is raised; *sig* is passed 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 entered
- **SIGSEGV**(5) when a segmentation exception is detected
- **SIGBUS**(7) when a bus error is detected

slot_address Routine

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

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

sprintf Routine

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

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

strcat Routine

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

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

strcmp Routine

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

This function compares successive characters from two strings, ***s1*** and ***s2***, until it finds characters that are not equal. If all characters are equal, **strcmp** returns 0. If different characters are found and the character from ***s1*** is greater than that from ***s2***, **strcmp** returns a positive number. If different characters are found and the character from ***s1*** is less than that from ***s2***, **strcmp** returns a negative number. (Both character strings are assumed to be of type **unsigned char**).

strcpy Routine

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

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

strlen Routine

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

This function returns the number of characters in string ***s1***. The terminating null character of ***s1*** is not included in the returned number.

strncat Routine

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

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

strncmp Routine

```
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, **strncmp** returns 0. If different characters are found and the character from ***s1*** is greater than that from ***s2***, **strncmp** returns a positive number. If different characters are found and the character from ***s1*** is less than that from ***s2***, **strncmp** returns a negative number. (Both character strings are assumed to be of type **unsigned char**).

strncpy Routine

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

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

strtol Routine

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

This function converts the initial characters of the string ***s*** to value type **long**. If the value of ***endptr*** is not **NULL**, the pointer to the unconverted portion of ***s*** is stored in ****endptr***. If ***base*** is 0, 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 **strtol** returns a value of 0.

testintr Routine

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

This function checks for an interrupt pending from a TURBOchannel option module in slot ***sn***. If an interrupt is pending, **testintr** returns 1. If no interrupt is pending, **testintr** returns 0. Because REX runs with all interrupts masked, this function indicates only 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.

test_parity Routine (TCF1)

```
int test_parity(int sn);
```

This function returns whether or not the System TURBOchannel interface has detected a parity error on addresses/data received from slot **sn**. The routine **test_parity** returns 0 if no parity errors have been detected, a negative number if bad parity has been detected. System firmware must clear any parity error indication *that might be latched in the system TURBOchannel interface* after testing the parity error bit.

time Routine

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

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

unsetenv Routine

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

This function removes ***name*** from the environment table and returns 0. If ***name*** is not defined, **unsetenv** returns a nonzero value.

wbflush Routine

```
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 (1-dimensional array) of pointers to routines. Table 5-2 lists the contents of the callback vector.

Table 5-2. 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)( );
b8	int	(*disable_parity)( ); (TCF1)
8c	int	(*enableintr)( );
b0	int	(*enable_block_mode)( ); (TCF1)
b4	int	(*enable_parity)( ); (TCF1)
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)( );
c0	void	(*(*malloc)( );) (TCF1)
00	void	*(*memcpy)( );
04	void	*(*memset)( );
74	void	(*msdelay)( );
30	int	(*printf)( );
2c	int	(*puts)( );
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)( );

(continued on next page)

Table 5-2 (Cont.). Callback Vector Contents

Offset	typedef	Description
10	char	*(*strcpy)( );
14	int	(*strlen)( );
18	char	*(*strncat)( );
20	int	(*strncmp)( );
1c	char	*(*strncpy)( );
3c	long	(*strtol)( );
90	int	(*testintr)( );
bc	int	(*test_parity)( ); (TCF1)
48	long	(*time)( );
68	int	(*unsetenv)( );
70	void	(*wbflush)( );
94	void	*Private; pointer to system-specific data
c4 to d4		reserved

6 Firmware Revision Descriptions

TCF0

Definition of TCF0 System Firmware.

TCF0 is defined to be anything prior to TCF1. The getsysid( ) callback is used by Option Firmware to determine the System Firmware Revision level.

TCF1

Definition of TCF1 System Firmware.

This section describes the callbacks available in the TCF1 firmware revision, considerations for maintaining TCF0 compatibility in option firmware, and coding examples for using some of the callbacks which are new in TCF1.

Callbacks which are provided in TCF1 firmware

Note that some of the callbacks are not intended for option use and should be avoided. Callbacks labeled as "NOT FOR OPTION USE" or "Options should not use" return an error status in TCF1 system firmware implementations.

Table 6-1. Callbacks in TCF1 Firmware

RCV Offset	Function	Notes
0x54	bootinit( )	Address provided by Option boot function
0x58	bootread( )	Address provided by Option boot function
0x5c	bootwrite( )	Address provided by Option boot function
0x7c	clear_cache( )	
0x98	console_init( )	Options should not use, returns a negative value
0x88	disableintr( )	
0xb8	disable_ parity( )	New in TCF1, turn off system side parity checking
0x8c	enableintr( )	Enable testintr( ) for this slot
0xb0	enable_block_ mode( )	New in TCF1, enable system block mode I/O writes
0xb4	enable_ parity( )	New in TCF1, turn on system side parity checking
0xa8	execute_cmd( )	NOT FOR OPTION USE, returns a negative value
0x84	getbitmap( )	Option should not use, returns 0
0x24	getchar( )	
0x64	getenv( )	For REX standard environment variables only
0x28	gets( )	
0x80	getsysid( )	
0xa4	gettcinfo( )	
0x9c	halt( )	Options should not use, does nothing
0x38	io_poll( )	
0x78	leds( )	For debug only, response is system dependent
0x50	longjmp( )	
0xc0	malloc( )	New in TCF1, allocate buffer space

(continued on next page)

Table 6-1 (Cont.). Callbacks in TCF1 Firmware

RCV Offset	Function	Notes
0x00	memcpy( )	
0x04	memset( )	
0x74	msdelay( )	
0x30	printf( )	
0x94	private( )	NOT FOR OPTION USE, returns a negative value
0x2c	puts( )	
0x44	raise( )	Response is system dependent
0xac	rex	Options should not use, does nothing
0x60	setenv( )	Options should not use, returns a negative value
0x4c	setjmp( )	
0xa0	showfault( )	Options should not use, does nothing
0x40	signal( )	Response is system dependent
0x6c	slot_address( )	
0x34	sprintf( )	
0x08	strcat( )	
0x0c	strcmp( )	
0x10	strcpy( )	
0x14	strlen( )	
0x18	strncat( )	
0x20	strncmp( )	
0x1c	strncpy( )	
0x3c	strtol( )	
0x90	testintr( )	
0xbc	test_parity( )	New in TCF1, test good/bad parity at system TURBOchannel interface for last transfer
0x48	time( )	
0x68	unsetenv( )	Options should not use, returns a negative value
0x70	wbflush( )	
0xc4-0xd4	reserved	

Maintaining Compatibility with TCF0 System Firmware

When a TCF1 option is plugged into a system which supports TCF0 firmware, it is strongly recommended that the option firmware remain TCF0-compatible by skipping any code that requires TCF1 functionality. Options must have the "TCF0" (TCF-Zero) string in their ROM header 'Firmware Type' field to remain compatible with *existing TCF0 System Firmware, regardless of whether or not the option uses TCF1 functionality*. For example, parity tests and parity enabling should be skipped if option firmware gets back a TCF0 code from getsysid( ) on a given system. This will allow new (TCF1) options to be compatible with existing (TCF0) systems.

Options must call the getsysid( ) callback and examine byte 1 to ensure the *system firmware* supports TCF1 before attempting to use any TCF1 functionality. Table 6-2 shows whether an option is compatible with TCF0 System Firmware or not. The first row corresponds to existing Option and System implementations, and the second row to TCF1 options which are remaining compatible with TCF0 systems. The situations in the third and fourth rows depict an option which must have TCF1 system firmware capability in order to run properly, these two cases should be avoided wherever possible.

Table 6-2. System Firmware Compatibility

Option ROM Header 'Firmware Type' Field	System Firmware Revision	Level of Option Compatibility
"TCF0"	TCF0 (=2)	Option Firmware may use only TCF0 features
"TCF0"	TCF1 (=3)	Option firmware may use TCF1 features on this system. On TCF0 systems, option firmware will be compatible if no TCF1 callbacks are used.
"TCF1"	TCF0 (=2)	Option not compatible with this system.
"TCF1"	TCF1 (=3)	Option is compatible with this system, but not with TCF0 systems.

Coding Examples

The examples below are provided only to show the flow and structure of code necessary to utilize TCF1 features.

The examples are not meant to be complete. They are provided only to show the required flow for maintaining TCF0 firmware compatibility when a TCF1 option is plugged into a system supporting only TCF0 firmware. If an option *must* have TCF1 functionality in order to properly function, then it should print an error message and exit with severe error status (-2) if it detects TCF0 in the getsysid( ) return code. The TCF0 System Firmware Response to an Option with "TCF1" in its header is *system dependent*.

In the examples that follow, TCF1 is used in place of the value 0x03, which is returned in byte 1 of the value returned by getsysid( ).

Example 1: *Maintaining TCF0 Compatibility*

Options which implement parity will be of type TCF1. However, it is *strongly recommended* that options which implement parity remain TCF0 compatible. This is done by adapting the following code sequence into the option firmware:

```
routine_entry(int argc, char **argv, int slot, rcv *rcv_ptr)
{
   int sys_id;
   sys_id = getsysid();
  if ( ((sys_id >> 8) & 0xff) >= TCF1)
  {
      /*System firmware supports TCF1 functionality.*/
      /*CALLS TO TCF1-ONLY CODE GO HERE (e.g., parity tests)*/
  }
 /*CALLS TO TCF0-COMPATIBLE CODE GO HERE*/
}
```

Example 2: *Good Parity Test*

A good-parity test (i.e., make sure the system is not detecting parity errors on the option slot) should follow the sequence:

```
 :
if (enable_parity(slot) ==0) /*Turn on system parity detection*/
{
   data_val = *(some_option_io_address);
   if (test_parity(slot) ! = 0
   {
      /*ERROR - parity error detected, expected good parity.*/
   }
}
else
{
     /*ERROR - system could not enable parity*/
}
 :
```

Example 3: *Bad Parity Test*

A bad-parity test (i.e., make sure the host TURBOchannel interface will detect a parity error in the event one should occur) should follow the sequence:

```
 :
if (enable_parity(slot) == 0)   /*Turn on system parity*/
{
   /*Code to set up option to generate bad parity goes here.*/
   data_val = *(some_option_io_address);
   if  (test_parity(slot) == 0)
   {
     /*ERROR - good parity detected, expected bad parity. */
   }
}
else
{
  /*System could not enable parity if here.*/
}
 :
```

Example 4: *Enabling Parity*

If parity is to be enabled, option firmware *must* perform the following sequence. The sequence *must* appear in the *init* function (init object), and may in addition, *optionally* appear in other ROM objects.

```
if ( ((getsysid() >> 8) & 0xff) >= TCF1)  /*TCF1 == 3*/
{
     if (enable_parity(slot) == 0)
     {
/*Parity now enabled on host side, enable parity on the option.*/
        /*CODE TO WRITE OPTION SPECIFIC CSR GOES HERE*/
     }
     else
     {
       /*System could not enable parity if here.*/
     }
}
```

Example 5: *Enabling Block I/O Writes*

If Block I/O Writes are to be enabled, option firmware *must* perform the following sequence. The sequence *must* appear in the *init* function, and may in addition, *optionally* appear in other ROM objects.

```
 ⋮
if ( ((getsysid() >> 8) & 0xff) >= TCF1)  /*TCF1 == 3*/
{
    if ( (blk_size = enable_block_mode(slot, n_words)) > 0)
    {
    /*Block mode now enabled for 'blk_size' sized transfers.*/
    }
}
 ⋮
```