CONVEX Processor Diagnostics Manual (C3400 Series)

First Edition

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

CONVEX Processor Diagnostics Manual (C3400 Series)

Order No. DHW-302

First Edition
May 1992

CONVEX Press
Richardson, Texas
United States of America

CONVEX Processor Diagnostics Manual (C3400 Series)

Order No. DHW-302

Printed in the United States of America

Revision information for

CONVEX Processor Diagnostics Manual (C3400 Series)

Edition	Document No.	Description
First	760-003830-000	Released May 1992. Material for this manual was taken from *CONVEX Processor Diagnostics Manual (C200 Series)* and *CONVEX Diagnostic Utilities Manual (C200 Series)*. Additional material has been added for functions specific to the C3400 Series supercomputers.

Contents

Figures

Tables

How to use this manual

The *CONVEX Processor Diagnostics Manual (C3400 Series)* documents the service processor-based processor diagnostics for CONVEX C3400 Series supercomputers and is intended to be the primary source of information on how to use these diagnostics.

This manual is not a tutorial, but rather a reference for field service and manufacturing test personnel, as well as CONVEX customers that perform their own system maintenance.

Test programs described in this manual are for:

- Central processing unit (CPU)
- Service processor unit (SPU)
- Main memory
- Peripheral interface adapter (PI2)

I/O and peripheral test programs are documented in the *CONVEX PBUS I/O System Diagnostics Manual.*

Organization

Each chapter in this manual covers a specific diagnostic function. Test descriptions and the invocation procedures are included with each test.

- **Chapter 1, "Diagnostic environment—hardware"**— Contains an introduction to the hardware facilities and concepts that underlie processor diagnostics.
- **Chapter 2, "Diagnostic environment—software"**— Contains an introduction to the system software and concepts that underlie processor diagnostics.
- **Chapter 3, "Service processor EPROM-based self-tests"**—Describes the service processor EPROM-based diagnostic tests.
- **Chapter 4, "Service processor peripheral tests (spu2000)"**—Describes the service processor peripheral diagnostic tests.
- **Chapter 5, "Diagnostic utilities"**—Contains the instruction pages for the individual diagnostic utilities.
- **Chapter 6, "Diagnostic file formats"**—Contains the formats of system files commonly use by the diagnostic utilities.
- **Chapter 7, "Diagnostic shell (dshell)"**—Provides an overview of the dshell and `dshell` commands.
- **Chapter 8, "Service processor interface tests (spu4000)"**—Describes the service processor interface diagnostic tests.
- **Chapter 9, "Memory subsystem diagnostic tests (mem4100)"**—Describes the memory subsystem diagnostic tests.
- **Chapter 10, "PI2 functional tests (`pi2_4000`)"**— Describes the peripheral interface adapter (PI2) functional diagnostic tests.
- **Chapter 11, "Scalar building block tests (`cpu4030`)"**— Describes the scalar instruction tests.
- **Chapter 12, "Vector instruction tests (`cpu4041`)"**— Describes the vector processor unit interface diagnostic tests.
- **Chapter 13, "Enhanced vector instruction tests (`cpu4241`)"**—Describes the CPU-specific vector diagnostic tests.
- **Chapter 14, "Privileged instructions and architectural features (`cpu4331`)"**—Describes the diagnostic tests specific to the C3400 Series processors.

- **Chapter 15, "Enhanced, nonvector uniprocessor instructions (`cpu4332`)"**—Describes the CPU-specific nonvector diagnostic tests.
- **Chapter 16, "Multiprocessor diagnostics (`cpu4333`)"**—Describes the CPU-specific multiprocessor diagnostic tests.

Notational conventions

Notational conventions are those characters, symbols, terminology, or abbreviated expressions used in this guide.

Text notation

Text notation conventions set apart special items in text or examples.

- `Monospace type`, represents computer output, binary and hexadecimal numbers, commands, instructions or mnemonics, data fields.

  Example:

  ```
  ERROR: Unknown command. Reenter.
  ```

- **`Bold monospace type`**, represents your response to a program or utility prompt.

  Example:

  ```
  Do you really want to exit? y
  ```

- Bold uppercase names designate keycap names.

  Example:

  RETURN

- If two keycap names are separated by a space, they are to be pressed sequentially.

  Example:

  ESC Q

- If two keycap names are separated by a hyphen, they are to be pressed simultaneously.

  Example:

  CTRL-C

- The word "enter," followed by a command, means to type the command and then press **RETURN**.

- *Italicized words* in an example command sequence are representative and are to be replaced with a user-supplied name, such as a file name.

  Example:

  `command` *filename*

- Angle brackets (< >) designate unprintable ASCII characters.

  Example:

  `<197>` is an em dash

- Angle brackets (< >) are used to designate fields as bits in a byte, word, register, and so forth.

  Example:

  `PSW <6...0>`

- Square brackets ([ ]) in a command sequence designate optional letters, characters, subcommands or other command elements. Brackets may be nested, indicating optional subelements. If there are two or more options they are separated by vertical slashes or pipe symbols.

  Example:

  `com[mand]` [*filename*|*devicename*]

- Braces ({ }) in a command sequence designate mandatory input, which must be one of two or more possible options. These options are separated by vertical slashes or pipe symbols.

  Example:

  `com[mand] {a|b|c}`

- A vertical slash (|), also known as the pipe symbol, in a command sequence indicates "or," giving you a choice between optional elements of a command.

  Example:

  `conf[igure]` [*command* | *alias*]

- Horizontal ellipses (. . .) in a command sequence show that the element immediately preceding them can be repeated.

  Example:

  ```
  ad[d] [ [board] . . . | all]
  ```

- Vertical ellipses in a command sequence show that lines of an example have been left out.

  Example:

  ```
  Verifying image   99
  Verifying image  199
  .
  .
  .
  Verifying image  999
  ```

Command syntax

Notational conventions above are used to define the commands in the user interface, such as in the following example:

`com[mand]{.t|.f} [-a|-b]` *input_file* `[...]` [*output_file*]

Where:

- `command` is *required* and may be abbreviated to `com` (square brackets indicate optional portion).
- If a command option, indicted by a list in braces, separated by a vertical slash, is used, then either `.t` or `.f` *is required.*
- If a command option, indicted by a list in square brackets, separated by a vertical slash, is used, then either `-a` or `-b` *is optional.*
- *input_file,* indicated by italics with no square brackets, is a *required* file name *supplied by the user.*
- Additional *input_file* names, indicated by ellipses in square brackets, *may optionally be supplied* by the user.
- *output_file,* indicated by square brackets and italics, is an *optional* file name *supplied by the user.*

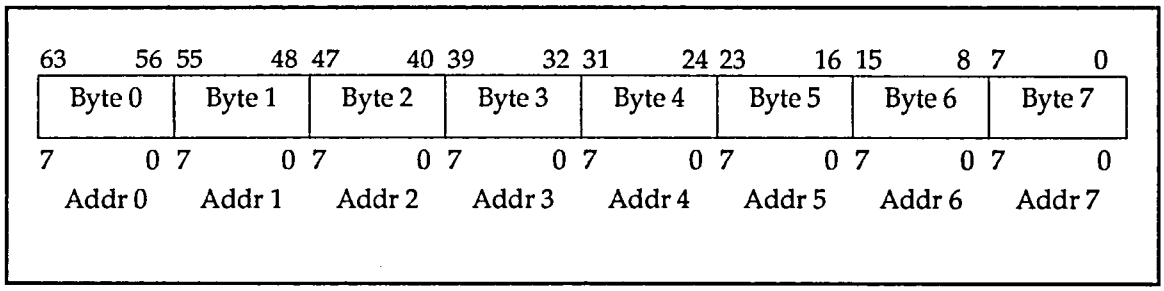
Data notation

Data notation conventions identify specific definitions in CONVEX supercomputer architecture:

- A *bit* is a single binary value or entity.
- A *nibble* is 4 bits.
- A *byte* is 8 bits.
- A *halfword* is 16 bits.
- A *word* is 32 bits.
- A *longword* is 64 bits.
- *Single precision* is a 32-bit floating point word.
- *Double precision* is a 64-bit floating point longword.
- An *instruction* is a multi-halfword operand.
- A bit is *set* when it contains a binary value of `1`.
- A bit is *clear* when it contains a binary value of `0`.
- Bit numbering is left to right, N-1 through 0. The most significant numerical bit is N-1, the least significant 0. The bit numbering represents the binary weight of a position.
- Byte numbering is from left to right, 0 through N-1.
- Byte order in a 64-bit longword is interpreted with increasing byte addresses associated with higher order bytes within a longword. The most significant bit is associated with the least significant byte number.

  Figure 1 represents the ordering of each addressable entity within a 64-bit longword.

Figure 1
Memory longword structure



- Bit fields are specified as:

  ```
  reg_name<x..y>
  ```

  where the bit field is `reg_name` from bits `x` through `y`.
- Individual bit positions within a register are specified as:

  ```
  reg_name<15,4,0>
  ```

  where 15, 4, and 0 are bits within `reg_name`.
- A *register* is a programmer-visible hardware storage element internal to the CPU.
- All register contents are written in hexadecimal notation unless explicitly stated otherwise.
- Individual bit positions within a register are denoted by specific positions separated by commas. For example, Vk<15,4,0> denotes bits 15, 4, and 0 of Vk.
- Individual vector element positions within a vector register are denoted by specific positions separated by commas. For example, Vk(15,4,0) denotes elements 15, 4, and 0 of Vk.
- An *instruction* is a group of halfwords. For C100 Series CPUs, the first halfword is an op code and the remaining halfwords are operands. For C200 Series CPUs, the first halfword is a op code prefix, one halfword is an op code, and the remaining halfwords are operands.
- All memory and I/O addresses are written in hexadecimal notation unless explicitly stated otherwise.
- *Physical memory* is the physical storage installed in the CPU.
- *Virtual memory* is the perceived amount of main memory as seen by the application programmer.
- The symbol *K* is an abbreviation for *kilo* or 1,024.
- The symbol *M* is an abbreviation for *mega* or 1,048,576.
- The symbol *G* is an abbreviation for *giga* or 1,073,741,824.
- A *stack* is a data structure in which memory is allocated and deallocated from one end, usually called the top, on a last-in-first-out basis.
- A *return block* is a collection of register contents that are pushed on/popped off a stack in response to an instruction or other event.

- *Reserved* or *undefined* indicate what, if anything, to expect from unused fields in registers, reserved memory, or reserved I/O space. Algorithm implementation based on the use of reserved fields is not recommended.

Warnings, cautions, and notes

The following are examples of warnings, cautions, and notes, and their typical content and location, as used in CONVEX documents:

Warning

A warning highlights procedures or information necessary to avoid injury to personnel. The warning immediately precedes the critical information and includes a description of the hazard.

Caution

A caution highlights procedures or information necessary to avoid damage to equipment, damage to software, loss of data, or that leads to invalid test results. The caution immediately precedes the critical information and includes a description of the possible damage.

Note

A note highlights information of a supplemental nature. The note immediately precedes or follows the highlighted information.

Associated documents

The following is a partial list of other manuals or books that may provide more detailed information on the topics presented in this manual:

- *CONVEX SPU OS Utilities Manual*, Order No. DHW-007
- *CONVEX PBUS I/O System Diagnostics Manual*, Order No. DHW-008
- *CONVEX Architecture Reference Manual (C Series)*, Order No. DHW-300
- *CONVEX Assembly Language Reference Manual (C Series)*, Order No. DHW-301
- *CONVEX Assembly Language Timing Guide (C Series)*, Order No. DHW-303
- *CONVEX UNIX Tutorial Papers*, Order No. DSW-002

Ordering documents

To order the current edition of this or any other CONVEX document, send requests to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

Include the order number or exact title with the request. The order number is on the title page of the manual and begins with the letters "DHW-" or "DSW-."

The order number for the *CONVEX Processor Diagnostics Manual (C3400 Series)* is DHW-302.

Technical assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC):

- From all locations in the continental United States
  - CONVEX customers call (800)952-0379.
  - CONVEX employees call (800)545-4839.
- From locations in Canada, call (800)345-2394.
- From all other locations, contact the nearest CONVEX office.

Acknowledgments

Many people have made contributions to this document:

- Technical contributors: Brian Allison, Tom Ford, Steve Gardner, Tony Jones, Dave Rotheroe, Jeff Venters
- Document review team: Dan Brenner, Frank Chaney, Steve Fieler, Tom Ford, Jeff Gruger, Mark Jones, Chris Magargee, Rick Miller, David Muir, Craig Reed, David Rotheroe, Chip Stroup
- Hardware Documentation staff: Phil Burkett, Randall Stiles
- Illustrations: Josie Davis
- Editorial Services: Larry Bonura, Sheri Roloff

Without the efforts of all, this document would not have been possible.

Phil Lloyd
Processor Documentation Writer

Diagnostic environment—hardware

1

The CONVEX diagnostic environment is comprised of:

- Hardware components:
  - Service processor unit (SP5)
  - Bus system
  - Scan ring architecture
- Software components:
  - SPU OS
  - Diagnostic utilities
  - Diagnostic test programs

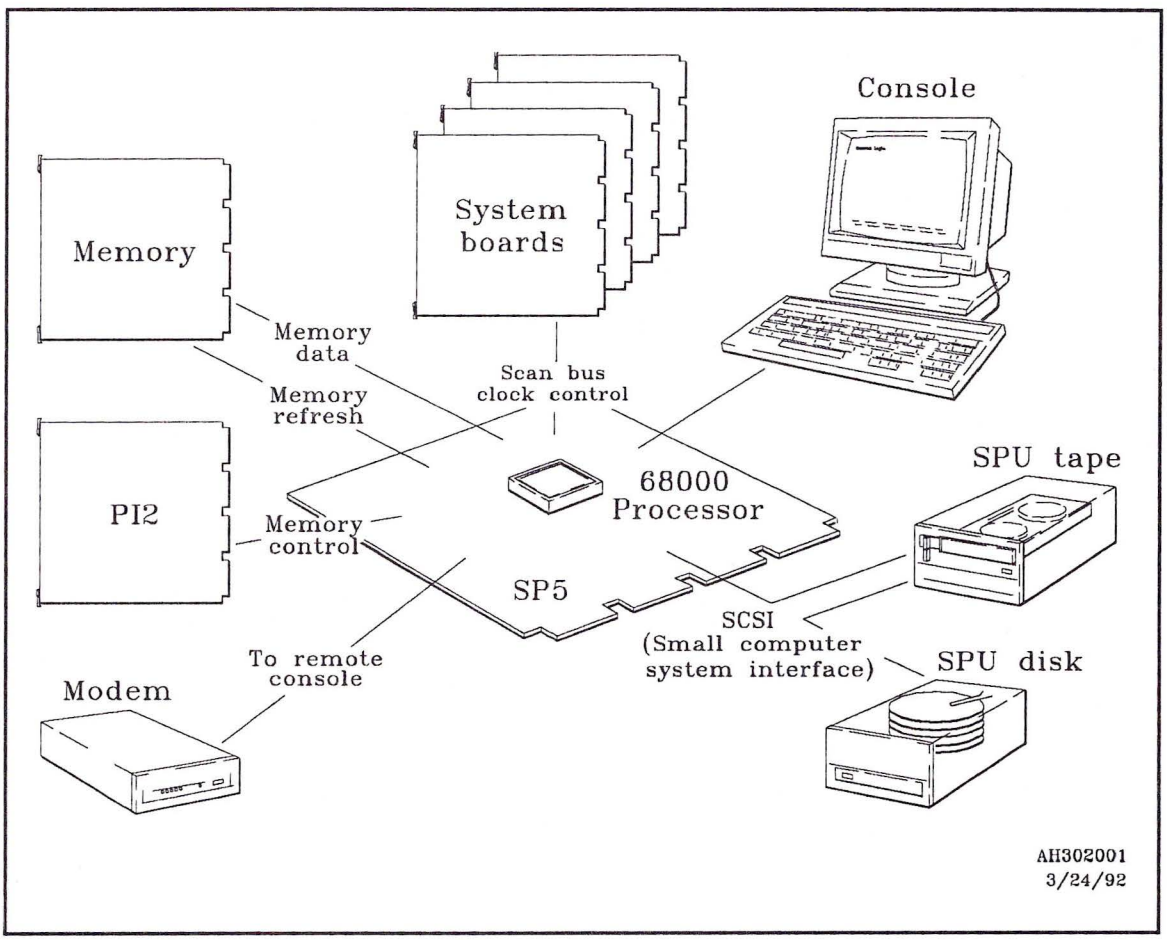
This chapter describes the hardware components of this diagnostic environment and provides background necessary to fully utilize the capabilities of the CONVEX processor diagnostics.

Hardware overview

Each CONVEX computer system contains built-in test hardware designed for system troubleshooting. The test hardware consists of a single-board microcomputer called the service processor unit (SPU) and a number of serial scan buses that permit the service processor to access the internal state of the system.

The basic features of the service processor hardware are shown in Figure 2.

Figure 2
Basic hardware features



Service processor unit

The service processor unit (SPU) for C3400 Series complexes is named SP5. The SPU is capable of operating autonomously with minimum support from other boards on the system.

The SPU consists of the following:

- Motorola 68000 microprocessor
- 64 kbytes of erasable programmable read-only memory (EPROM)
- 8 Mbytes of random-access memory (RAM)
- Two asynchronous RS-232 serial ports, one DCE and one DTE
- Small computer system interface (SCSI) bus
- 535-Mbyte, 3.5-inch hard disk
- 125-Mbyte, QIC-120 format quarter-inch cartridge tape drive

One of the asynchronous serial ports supports the system console, and the other is used to support execution of diagnostics from a remote site. The hard disk and cartridge tape drive are connected through the SCSI bus.

The SPU controls all the clocks within the systems by controlling the run signals that are sent from the SPU to every board in the system. In addition to clock control, the SPU also communicates with the rest of the system through three main bus structures:

- EBUS
- Interrupt bus
- Scan bus

The following sections describe these buses.

EBUS

The *EBUS* connects the SPU to the memory system. The SPU has the ability to do a variety of memory operations on the main memory system. The SPU has a set of map registers that map the portions of the SPU logical address space to access main memory. There are enough of these map registers to allow the SPU access to 4 Mbytes of memory at one time.

Interrupt bus

The *interrupt bus* communicates between the SPU and each CPU and channel control unit (CCU) within the system. Interrupts are used extensively during the diagnostics for communication between the SPU and the other processors within the system. When the SPU sends an interrupt, any subsystem listening for that interrupt will receive it.

There are 256 interrupts available within the system. The SPU has the ability to send all 256 interrupts and to receive interrupts 8 through 15.

Scan bus

The *scan bus* is really a set of serial buses connected to each board in the system. The scan bus permits the SPU to access and initialize the internal state of a system.

Each board in the system has one or more *scan rings*, a series of registers and register-like structures connected together serially. Each register has the ability to operate in either a parallel or serial load mode.

When the SPU is in a diagnostic mode, these rings operate in the serial load mode and appear to the SPU as a large shift register. The SPU controls these scan rings. The SPU can load and examine the state of any board in the system by writing and reading the scan rings.

Scan rings are used extensively during system initialization for board initialization and control store loading, and are used by test programs and diagnostic utilities for internal state monitoring and manipulation.

Board identification

Each board in the CPU chassis contains a serial electrically-erasable programmable read-only memory (EEPROM) chip that is accessible by the service processor via the scan bus. The EEPROM, frequently referred to as a COP chip, stores a variety of board identification information including part number, fabrication revision, and assembly revision. This information is used by the SPU to identify the scan ring configuration of a given board and to determine which boards are present in the CPU chassis.

Remote diagnostics port

In order to minimize hardware troubleshooting and problem isolation time, remote diagnostics capability is designed into the SPU. With a modem connected to the second (remote) asynchronous port on the SPU, it is possible for CONVEX Technical Assistance Center (TAC) personnel to execute diagnostics remotely. In this manner, it is frequently possible to identify the failing field replaceable unit (FRU) before dispatching field service personnel to the customer site.

Scan ring concept

A description of the scan ring concept begins with a definition of rings and fields, a diagram of scan ring bus architecture, and a description of scan ring field names.

A *field* is a combination of 1 to 32 bits. It is the smallest meaningful combination of bits. The bits in a field usually are pulled from a similar physical or functional area of a board. By inputting data into the bits of fields and monitoring the outputs, scanning acts as a valuable diagnostic tool.

A *scan ring*, or just *ring*, is a series of fields grouped together. A scan ring is comprised of fields that are always pulled from the same board. The fixed bit locations of a scan ring reside in serial shift registers, and are later shifted out of these registers when being scanned.

Ring names usually correspond to board types, and boards usually correspond to backplane slots. This is why ring names are sometimes referred to as *slots*. However, certain slots may house boards other than the board type they describe, and certain boards may contain more than one ring. These situations are the exception.

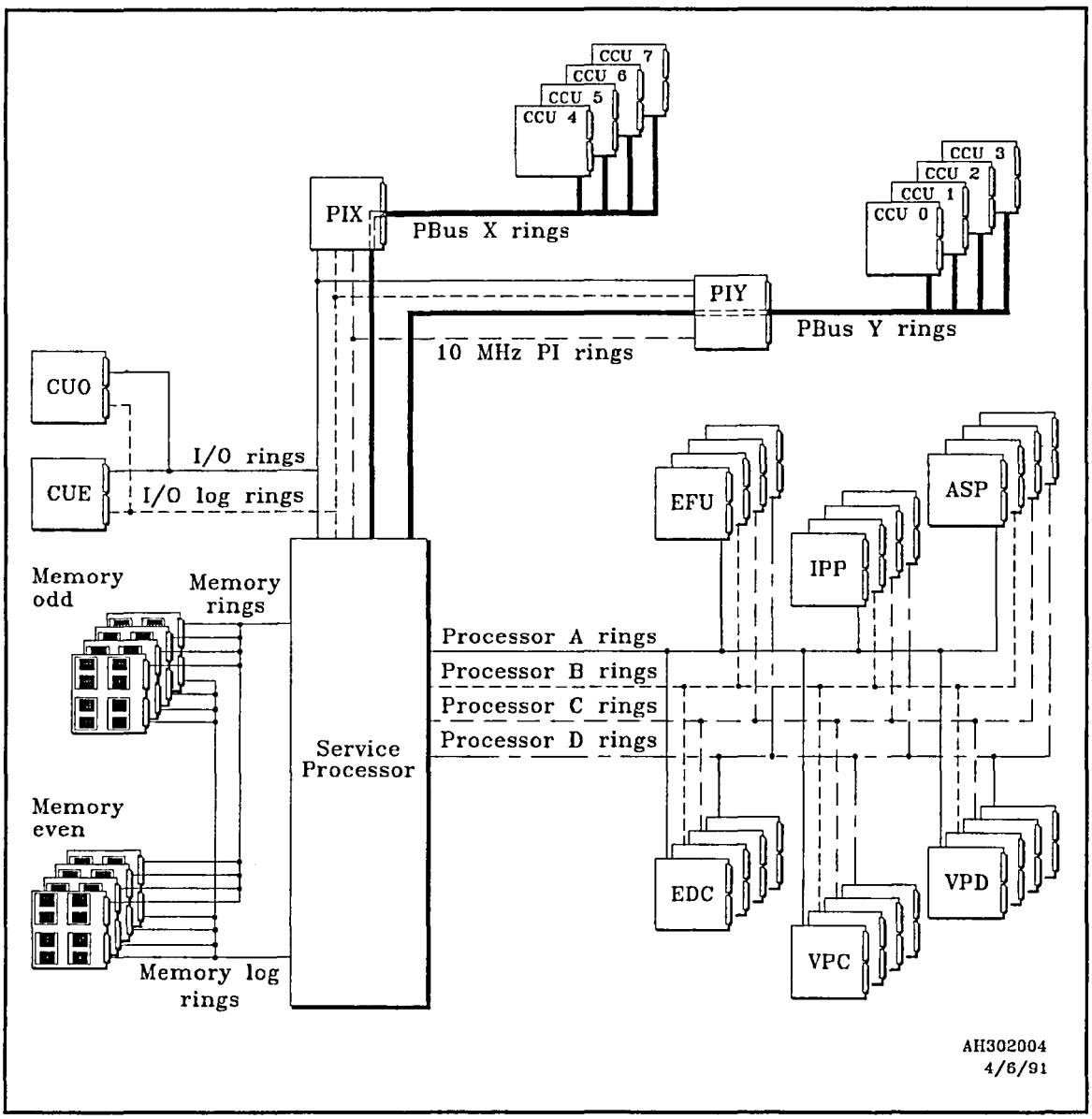
Scan ring architecture

The figures in this section illustrate the broad extent of the architecture used in scan rings and the buses used to carry scan information. They are helpful in understanding the scan rings available and their dependence on or independence from each other.

While different rings on the same board use the same backplane data path to communicate with the service processor, Figure 3 contains separate lines for each type of ring, in order to show they are independent.

An example of the dependence and independence illustrated in Figure 3 is the memory subsystem. While all normal and log memory scan rings use the same physical data path to get data from or send data to the service processor, the normal and log scan rings are independent of each other. A log scan ring may be scanned while the memory remains running, and a regular ring may be scanned without affecting the log functionality. All normal rings are dependent on each other, however, as are all log rings. To scan any normal ring, all other normal rings must be disabled. Likewise, to scan any log memory scan ring, all other log memory scan rings must be disabled.

Figure 3
Scan ring bus architecture

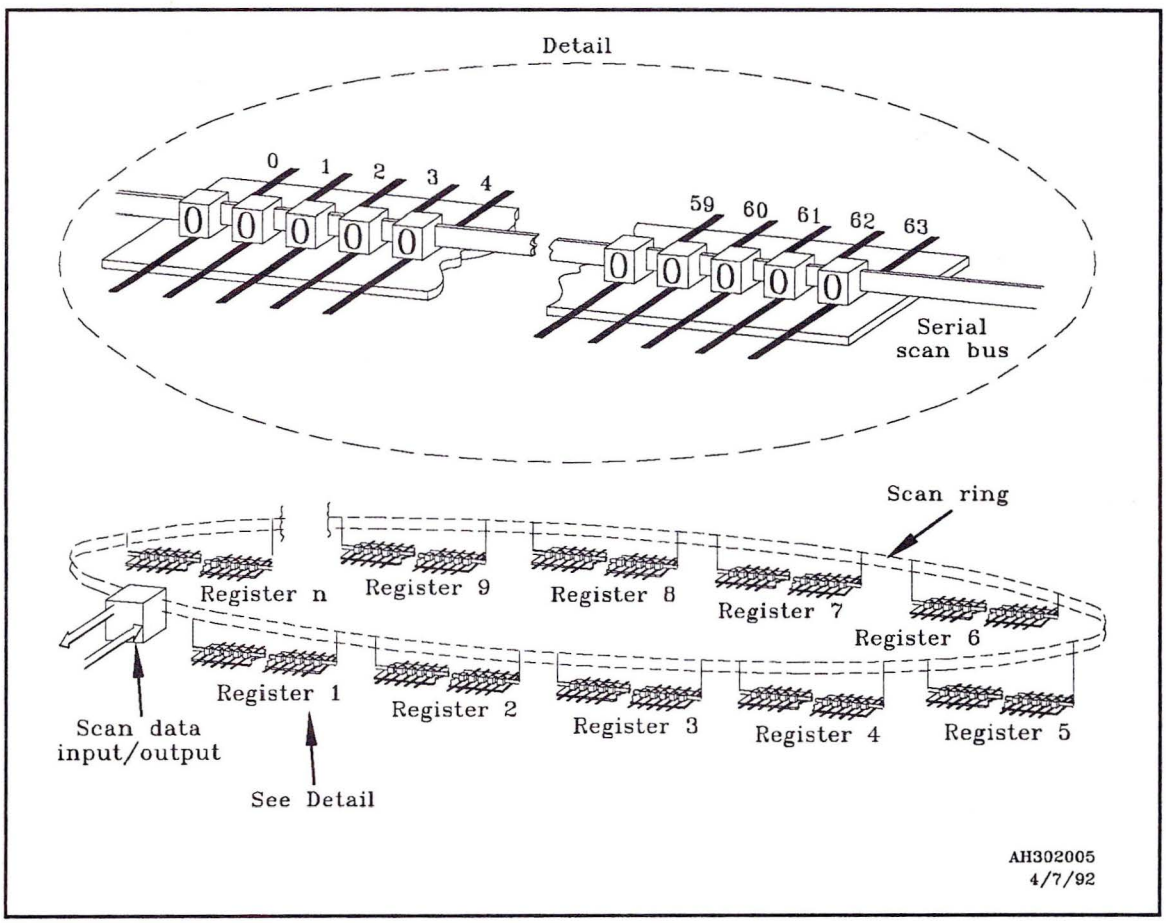


Scan rings

There are registers (and register-like structures) in the machine that normally operate in a parallel fashion but include a serial input and output connections. These registers also have the ability to shift data bits right or left on command. Portions of some rings are unidirectional; they shift in the *right* direction only.

A number of these registers are connected together with the serial output of one register connected to the serial input of the next. Finally, the serial output of the last register in the line is connected back around to the serial input of the first register. The serial configuration of these registers is then circular, creating a scan ring, as illustrated in Figure 4.

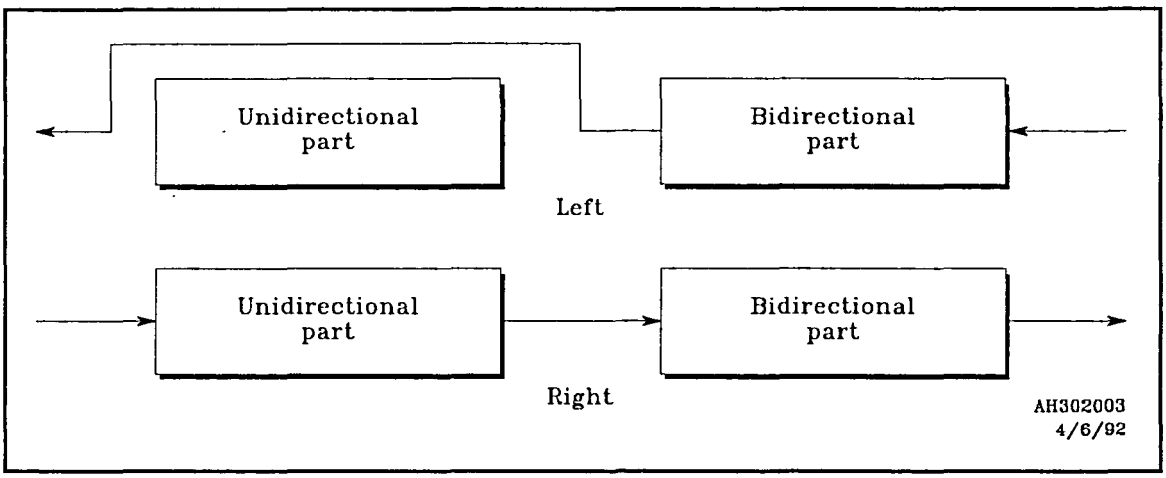
Figure 4
Scan ring



All scan rings may be thought of as being made up of two parts: a *bidirectional* part and a *unidirectional* part, each of which may have any length, including zero. The terms bidirectional and unidirectional come from the ability to scan that part of a scan ring. Bidirectional parts may be scanned in both the left and right directions. Unidirectional rings may be scanned only in the right direction, and are bypassed when scanning left. A scan ring that has both unidirectional and bidirectional parts will always have the bidirectional part be at the least significant end of the scan ring.

Figure 5 shows directional movement in a generic scan ring.

Figure 5
Scan ring directions



There are three types of scan rings:

- **Fully bidirectional**—The unidirectional part has zero length.
- **Partly bidirectional**—Both parts have nonzero lengths.
- **Fully unidirectional**—The bidirectional part has zero length.

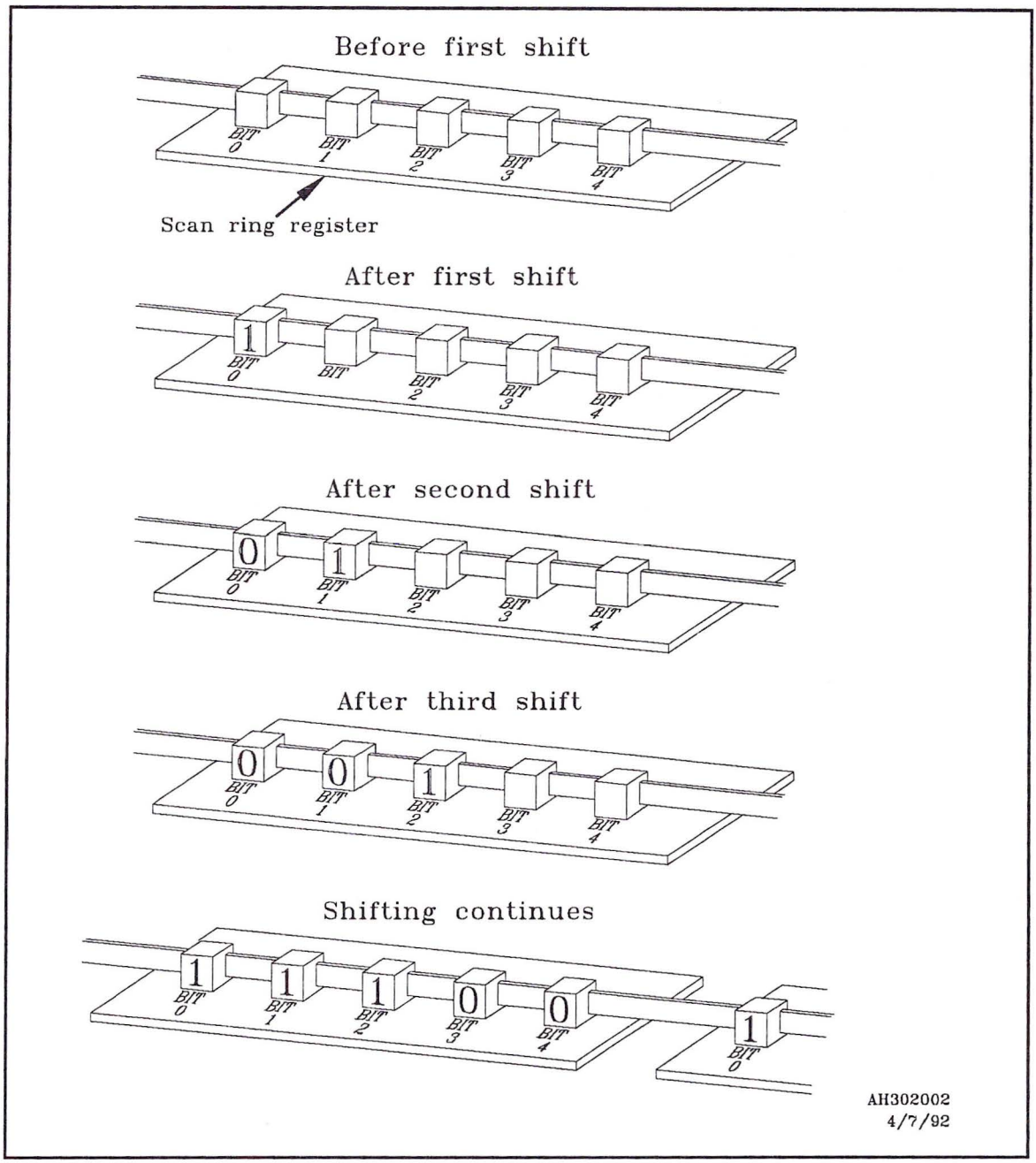
All fields of all scan rings may be read or written when scanning right. When scanning left, only the fields in the bidirectional portion of the scan rings are accessible.

The service processor controls the scan rings through its scan interface, which contains registers, scan control logic, and clock control logic. The other boards in the system are connected by lines to, and controlled by, the scan interface registers during scan operations.

Loading scan rings

Scan data is loaded into a ring sequentially, one bit at a time, as shown in Figure 6.

Figure 6
Loading scan ring registers



The first bit is applied to the serial input of the first register and then the registers are shifted. This moves the first bit into the first bit position of the first register. The next bit is then applied to the input of the first register and the registers shifted again (the same direction as before). This moves the first bit into the second position in the register and moves the second bit into the first register location.

The next shift moves everything over once again, with the third bit being moved into the first register location. When the first bit emerges from the serial output of the first register, it goes into the serial input next register in the ring on the next shift. This process is repeated until the entire scan ring is loaded with bits, with each located in its proper position.

In a similar fashion, a single bit can be placed in any location in the scan ring by applying the bit to the input and issuing the required number of shifts to move it around to the wanted location in the ring. If a particular scan register is 64 bits wide, it takes 65 shifts to get a bit into the register and then shifted out the other end. A scan ring may include hundreds of registers, and, therefore, thousands of bits.

Reading scan rings

Scan rings are read by shifting the registers in the ring until the wanted bit or bits come around to be read. Reading a scan ring is nondestructive so long as rings with unidirectional portions are not shifted in the *left* direction. The data remains in the ring after any number of right shifts during a read operation. Data can be restored to its original position in the ring by shifting the ring the appropriate number of times to move the data full circle back to its original location in the ring.

Scan ring names

Scan ring name definitions and the associated field name definitions are generated directly from a scan ring description file. The definitions of scan ring fields, their formats and ring specifications are inherent in the variable names used in the diagnostic database.

This method of defining the scan fields and associated information is required for compatibility with C language library calls.

Field names are defined by the `scan_builder` utility. The `scan_builder` is a program that creates the structures needed by C programs to access the scan rings. The `scan_builder` creates a collection of files that form the database for field and ring definitions. The `scan_builder` constructs a field name according to the pattern in Figure 7.

Figure 7
Scan ring name pattern

In Figure 7:

- *ringname* designates the scan ring to be accessed.

  This is not always synonymous with the board name. A board may have more than one scan ring associated with it. There may also be more than one board of the same name in a system.

- *field_def*[*index*] is a field definitions for the ring to be accessed.

  There is an optional index on each field definition and on the ring name.

The following are examples of how `iscn` field names must be constructed.

```
mcm[0]:wr_dat[2]
mcm[1]:xbar[1].is[0]
```

The first example specifies the `wr_dat[2]` field, in the memory array ring `mcm[0]`. The second example specifies the `xbar[1].is[0]` field, in the memory array ring `mcm[1]`. The last character in every ring name is always a colon.

Two `iscn` utility commands are useful in becoming familiar with scan ring and field names.

Entering `list` displays the name of all rings that are present in a particular configuration, and if used at system initiation may alert the user to board changes. The names of empty or nonexistent board slots are not displayed.

Board names usually describe the slot in which they fit. For example, the same board will always go in slot CUJ. In other cases, there is some variety in the type of boards that can go in a slot.

Entering `get`, followed by a ring name, displays the value of all the fields within a ring.

The following six tables list example ring names that are currently defined for C3400 Series computers.

Table 1
Service processor ring name definitions

Ring name	Alias
Service processor ring	SP5

Table 2
Central processor ring name definitions

	Aliases for processors							
Ring name	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7
CPU normal right shift ring	cpu[0]	cpu[1]	cpu[2]	cpu[3]	cpu[4]	cpu[5]	cpu[6]	cpu[7]
CPU complete right shift ring	cpur[0]	cpur[1]	cpur[2]	cpur[3]	cpur[4]	cpur[5]	cpur[6]	cpur[7]
CPU left shift ring	cpul[0]	cpul[1]	cpul[2]	cpul[3]	cpul[4]	cpul[5]	cpul[6]	cpul[7]

Table 3
Memory array ring name definitions

Ring name	Memory pairs for processors							
	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7
Memory array Run ring even	mcm[0] me0	mcm[2] me1	mcm[4] me2	mcm[6] me3	mcm[0] me0	mcm[2] me1	mcm[4] me2	mcm[6] me3
Memory array Run ring odd	mcm[1] mo0	mcm[3] mo1	mcm[5] mo2	mcm[7] mo3	mcm[1] mo0	mcm[3] mo1	mcm[5] mo2	mcm[7] mo3
Memory array Log ring even	lmcm[0] lme0	lmcm[2] lme1	lmcm[4] lme2	lmcm[6] lme3	lmcm[0] lme0	lmcm[2] lme1	lmcm[4] lme2	lmcm[6] lme3
Memory array Log ring odd	lmcm[1] lmo0	lmcm[3] lmo1	lmcm[5] lmo2	lmcm[7] lmo3	lmcm[1] lmo0	lmcm[3] lmo1	lmcm[5] lmo2	lmcm[7] lmo3

Table 4
CPU utility board ring name definitions

Ring name	Alias
CPU utility board rings	cuj
CPU utility board log rings	lcuj

Table 5
Channel control unit ring name definitions

Ring name	Aliases
Channel control unit rings	ccu[0], ccu[1], ... ccu[5]

Table 6
PBUS interface adapter ring name definitions

Ring name	Alias
PBUS interface adapter ring	pi[0]/piy, pi[1]/pix
PBUS interface adapter log ring	lpi[0]/lpiy, lpi[1]/lpix
PBUS interface adapter 10MHz ring	opi[0]/opiy, opi[1]/opix

Diagnostic environment—software

2

The CONVEX diagnostic environment is comprised of:

- Hardware components
  - Service processor
  - Bus system
  - Scan ring architecture
- Software components
  - SPU OS
  - Diagnostic utilities
  - Diagnostic test programs

This chapter describes the software components of the diagnostic environment and provides background necessary to fully utilize the capabilities of the CONVEX processor diagnostics.

CONVEX system diagnostics consist of a suite of utilities and various test programs designed to execute under the service processor operating system, SPU OS (except where noted). These programs use the capabilities of the service processor to test the operation of one or more of the system functions and report any errors detected.

All diagnostics in this chapter are intended to be executed off-line, that is, while ConvexOS is not being executed by any of the central processing units (CPUs) in the system.

Software overview

Software in the diagnostic environment is hierarchically structured as shown in Table 7.

Table 7
Hierarchical software structure

Level	Description
1	Soft front panel
2	SPU OS
3	Diagnostic utilities and diagnostic shell (dshell)
4	Diagnostic tests

Soft front panel

At the highest level of the hierarchy is the *soft front panel*. This is an EPROM-based program that permits specification of a variety of parameters related to system booting and operation. It also includes a series of EPROM-based self-tests. The SPU operating system, SPU OS, is booted from the soft front panel.

SPU OS

The second level of the hierarchy, *SPU OS*, manages the SPU memory and peripheral resources, and provides the environment for the execution of system diagnostics. The instruction set of SPU OS is less rich than ConvexOS, but the user interface is recognizable to anyone familiar with the UNIX Bourne shell.

Diagnostic utilities

The next level of the hierarchy consists of the *diagnostic utilities*. These utilities are programs developed to perform a variety of initialization and system monitoring tasks. Included in this list of programs are control store loaders, memory initialization and display tools, error loggers, and a diagnostic test executive called the *diagnostic shell* (dshell).

The dshell is a CONVEX-developed command language interpreter specifically designed for controlling test program execution and error reporting. The dshell permits the user to specify portions of the test programs to be executed, the level of detail in the error report, and what action to take on the occurrence of an error such as, log multiple failures, pause on failure, loop on subtest, and so on.

Diagnostic tests

At the lowest level of the hierarchy are the *diagnostic tests*. Each test program is designed to verify specific functions of the computer system or of a particular subsystem. The test programs use the diagnostic features of the service processor to control system clocking, load object modules into main memory or channel control unit (CCU) memory, or initiate execution of code by the CPU and monitor the results.

Each component of the diagnostic software environment hierarchy is discussed in more detail in the following sections.

Soft front panel

The diagnostic environment begins at the soft front panel. The soft front panel is a program that is executed from the service processor erasable programmable read-only memory (EPROM), whenever the system is powered up or reset via the **RESET** button. Its behavior is controlled by an electrically-erasable programmable read-only memory (EEPROM) that is on the service processor. The contents of this EEPROM, a form of nonvolatile memory, can be modified interactively from the soft front panel during a session.

Since the service processor stores the setting of the soft front panel options in nonvolatile memory, they are preserved even when the system is powered off.

The SPU EEPROM is organized into 32 storage locations, numbered 0 to 31, each of which is 16 bits wide. These locations are used to store front panel options plus other information such as the SPU assembly revision, part number, and so forth.

The soft front panel provides an interactive user interface that executes from the local system console. The soft front panel allows access to the SPU self-tests, the commands to configure the system boot mode, and the ability to invoke the SPU OS bootstrap program.

Starting the soft front panel

The soft front panel is entered when the keyswitch is turned to the **LOCAL MAINTENANCE** or **REMOTE** position. If the keyswitch is in the **REMOTE** position, input is taken from the remote port and echoed to both the remote port and system console. If the keyswitch is in the **SECURE EXECUTION** position and automatic reboot is enabled, the interactive soft front panel is bypassed and the boot procedure is invoked. If at any point while in the soft front panel, the keyswitch is placed in the **SECURE EXECUTION** position, the soft front panel does not accept input from the keyboard. The **RESET** button on the front panel is also disabled.

The soft front panel can be entered in any of four basic ways:

- When the system is powered up
- When the **RESET** button is pushed
- When the `/etc/reboot` or `/etc/fasthalt` commands are executed from SPU OS
- When the SPU OS watchdog timer expires (a result of a crashed SPU)

Soft front panel menu

Each time the SPU is reset by a power up condition or whenever the **RESET** button on the front panel is pressed, self-tests 1 and 2 always execute. If the `spu-self-test` option is enabled, self-tests 3 through F are also executed. To run all 15 self-tests (1 through F) requires less than three minutes.

Before each self-test is started, a hexadecimal digit identifying the current self-test is output to the system console. If a failure is detected during one of the ts self-tests, the SPU halts and an error message may be displayed on the system console. The last digit displayed on the first line indicates which self-test detected the error.

For example, if Self-test 6 fails, the system console shows `123456` on the first line, possibly followed by an error message.

After the self-tests have completed successfully, the soft front panel menu is displayed, followed by the soft front panel prompt, `(fp)>`.

From the soft front panel prompt, you can select various modes of operation (such as diagnostic or normal operating system), default boot device, automatic reboot options, and other options which control self-test execution. All fields on the soft front panel menu have optional values which can be set by the user, except `SPU type` and `processor`.

Figure 8 illustrates a sample soft front panel menu.

Figure 8
Soft front panel menu

```
123456789ABCDEF

CONVEX Front Panel - Version: 3.32 / CPU Class: 7 / CPU SN 32514
SPU type = SP5                      Processor = 68000
mode-of-operation = normal-os       boot-device = disk
location-of-bootstrap = default     power-up-reboot = enable
automatic-reboot = enable           spu-self-test = enable
test-flags = normal                 remote-port-bps = 1200
SCSI-power-up-delay = 0xA           user-flags = 0x0
(fp)>
```

In the event a defective SPU results in a hung system during a self-test, *ALL* self-tests are skipped the next time the soft front panel is invoked. This prevents a deadlock situation. Refer to Chapter 3, "Service processor EPROM-based self-tests," for more information on the self-tests.

For more information on starting up and shutting down the system, refer to the *CONVEX SPU System Manager's Guide* and the *CONVEX Processor Operation Guide (C3400 Series)*.

Keyboard input

While in the interactive soft front panel, some keys on the keyboard have special meaning. They can be used to edit the command line as defined in Table 8.

Table 8
Edit keys

Keystroke(s)	Meaning
BACKSPACE	Back up and erase previous character
DEL	
CTRL-H	
CTRL-U	Back up and erase all previous input
CTRL-C	Abort all previous input and start over

Alphabetic keys are case sensitive and should be entered in uppercase, as shown in the examples.

Soft front panel commands

The soft front panel provides several commands and options, as listed in Table 9.

Table 9
Soft front panel commands

Command	Description
`boot`	Load and invoke the disk or tape bootstrap.
`debug`	Invoke hardware debugger menu.
`display`	Display soft front panel.
`execute`	Run specified EPROM self-test(s) (in the range 0 through `0x0f`).
`help`	List soft front panel commands and options.
`menu`	Display a self-test list.
`preset`	Preset nonvolatile soft front panel options to a sane state.
`reset/quit`	Restart EPROM as if the RESET button were pressed.
`set`	Set a soft front panel option.

All commands that modify options stored in the EEPROM, perform a read verify of the written data.

boot

Load and invoke the disk or tape bootstrap

Syntax

`b[oot]`

Function

Read a program into local SPU DRAM and execute it. The program is assumed to be 16 kbytes in size. The source for the `boot` program ("disk" or "tape") is determined by the `boot-device` soft front panel option. Other selections specify the SCSI target ID number and are assumed to be disk type devices. Where the program is loaded from (on the boot device) is determined by the `location-of-bootstrap` soft front panel option. The EPROM allows selection of four different bootstrap locations. If the `test-flags` option is set to `verbose`, more information is printed out during the boot cycle.

Operation

Boot procedure:

1. Set up and enable SPU memory mapper.
   a. Copy the EPROM to physical address `0x10000`.
   b. Map 64 kbytes (16 pages) beginning at logical address `0x0` to physical address `0x10000` (the EPROM copy).
   c. Map 16 kbytes (4 pages) beginning at logical address `0x20000` to physical address `0x0` (target address of bootstrap).
   d. Map the remaining portion of first 2 Mbytes as "logical address equals physical address."
   e. Enable the memory mapper.
2. Check if this is the first boot after a power up condition. Delay if needed.
   a. Read `power-up-boot` flag from EEPROM chip. If set, read `SCSI-power-up-delay`. If nonzero, delay that many seconds and reset `power-up-boot` flag in EEPROM chip.
3. Assert `SCSI-reset` on interface.
4. Wait for `SCSI-device unit ready`.
   a. Attempt `SCSI-test-unit-ready` command every second until ready or until 30 seconds have elapsed. If not ready after 30 seconds, assert `SCSI reset` on interface and repeat `ready wait` loop up to 5 times.

5. Boot procedure specific to disk.
   a. Read disk block size from device. All I/O is scaled to match the reported block size.
   b. Read 16 kbytes (4 pages) from disk to logical address `0x20000` (physical 0) starting at disk block: $(1 + N) + 32N$, where N is the boot location (0 through 3). The actual starting block is scaled based on the reported device block size.
6. Boot procedure specific to tape.
   a. Rewind the tape.
   b. Read the tape header (directory block).
   c. Read the number of tape filemarks to skip from header.
   d. Skip to first copy of bootstrap.
   e. Read 16 kbytes (4 pages) from tape and write to logical address `0x20000` (physical 0). Repeat this step until the bootstrap copy (0 through 3) has been read.
7. Checksum the 16 kbyte bootstrap to ensure integrity.
8. Invoke bootstrap.
   a. Disable the SPU memory mapper.
   b. Map the first 2 Mbytes as "logical address equals physical address."
   c. Enable mapper and simulate 68000 reset (start bootstrap).

Note

Soft front panel options affecting command operation:

- `boot-device`
- `location-of-bootstrap`
- `mode-of-operation`
- `test-flags`

Errors

Any errors that occur during the boot cause the display of the bytes comprising the last executed SCSI command before the error.

If the error was reported from the SCSI device, four bytes of sense data from the `request sense` are displayed, as shown in Figure 9.

Figure 9
Error message from a SCSI device

```
Boot Device Error. SCSI command (6 bytes): 0A 00 00 00 00 00
SCSI sense bytes: 0x70000A00
Current SCSI state: bus-free
```

For a list of SCSI sense data, refer to the target device SCSI manual.

If the error was not reported from the SCSI device, the two byte software error code is displayed, as shown in Figure 10.

Figure 10
Error message from a non-SCSI device

```
Boot Device Error. SCSI command (6 bytes): 0A 00 00 00 00 00
EPROM error code: 0x0001
Current SCSI state: bus-free
```

Table 10 is a list of EPROM detected error codes.

Table 10
EPROM detected error codes

Error code	Description
0000	The target SCSI device reported device busy during command status phase.
0001	The target SCSI device failed to assert busy after selection, or dropped busy before the command phase was started.
0002	The target SCSI device failed to handshake for all of the bytes comprising the SCSI command block (typically 6 bytes).
0003	During the SCSI data-in phase an EPROM timeout occurred after waiting 60 seconds for a SCSI transfer request.
0004	During the SCSI data-out phase an EPROM timeout occurred after waiting 60 seconds for a SCSI transfer request.
0005	An EPROM timeout occurred after waiting 1 second for the target SCSI device to enter the message-in phase.
0006	An EPROM timeout occurred while waiting for target SCSI device to enter the status phase following the command phase. The timeout is 10 seconds, except for tape device rewinds and filemark skipping, which have a timeout of 190 seconds.
0008	An error occurred during a SCSI request-sense command following a previous command that reported a check condition during the SCSI status phase.
0009	The target SCSI device left the data-in or data-out phase before the entire data block was transferred.
000C	An EPROM timeout occurred after waiting 10 seconds for the target SCSI device to enter the bus-free (not busy) phase following a nondata transfer type command.
0010	During a boot from the tape device, the tape directory magic number, or some other information read from the tape, was out of the expected range.

debug
Invoke hardware debugger

Syntax

`de[bug]`

Function

Invoke the hardware debugger and display the debugger prompt `(dbg)`.

All the commands available in the debugger are described in the "Hardware debugger" section of chapter 3, "Service processor EPROM-based self-tests."

Example

Figure 11 shows the steps to invoke the hardware debugger.

Figure 11
Invoking the hardware debugger

```
(sfp)> debug
(dbg)>
```

display
Display soft front panel

Syntax

`d[isplay]`

Function

Display the soft front panel. The menu is identical to the menu displayed when the soft front panel is initiated, but without the first line showing the tests performed, and without the `(fp)>` prompt.

Example

Figure 12 illustrates the soft front panel display.

Figure 12
Example soft front panel display

```
(sfp)> d
CONVEX Front Panel - Version: 3.32 / CPU Class: 7 / CPU SN 32514
SPU type = SP5                          Processor = 68000
mode-of-operation = normal-os           boot-device = disk
location-of-bootstrap = default         power-up-reboot = enable
automatic-reboot = enable               spu-self-test = disable
test-flags = normal                     remote-port-bps = 1200
SCSI-power-up-delay = 0xA               user-flags = 0x0
(sfp)>
```

execute

Run specified EPROM self-tests (1 through F)

Syntax

e[xecute] *test-number*

Function

Run a single EPROM self-test.

If the `test-flags` option is set to `continuous` or `all`, the test loops until the **RESET** button is pressed.

If the test-flags option is set to `verbose` or `all`, the text description of the subtest is printed prior to the execution.

Example

Figure 13 illustrates the soft front panel display.

Figure 13
Example soft front panel display

```
(sfp)> e a
```

Note

Soft front panel options affecting command operation:

- `test-flags`

help

List soft front panel commands and options

Syntax

`h[elp] | ?`

Function

Display the soft front panel command list, as illustrated in Figure 14. To display the help screen, enter `h[elp]` or `?` at the `(fp)>` prompt.

Example

Figure 14 illustrates the soft front panel help screen.

Figure 14
Soft front panel help screen

```
(sfp)> help
Command             Description,
boot                Load and invoke system bootstrap.
debug               Invoke the hardware debugger.
execute hex-value   Execute specified self-test.
display             Display the current option settings.
menu                Display self-test menu.
preset state        Preset all options to a predetermined state.
set Value           Set to the specified Value.

Examples:
preset [ standard | alternate | diagnostic | install ]
set mode-of-operation = [ normal-os | alternate-os | diagnostic |
other ]
set boot-device = [ disk | tape | 0 | 3 | 4 | 5 | 6 | 7 ]
set location-of-bootstrap = [ default|1st-copy|2nd-copy|3rd-copy ]
set power-up-reboot = [ disable | enable ]
set automatic-reboot = [ disable | enable ]
set spu-self-test = [ disable | enable ]
set test-flags = [ normal | continuous | verbose | all ]
set remote-port-bps = [ 1200|300|600|110|2400|4800|9600|19200 ]
set SCSI-power-up-delay = hex number [0-ffff]
set user-flags = hex number [0-ff]
Commands, option names, and value names may be abbreviated.
Hex values should be entered with a leading 0 or numeric character.
(sfp)>
```

All numeric values entered in hexadecimal must begin with a numeric character. For example, `15` decimal is entered as `0F` hexadecimal.

Optional parts of each command are shown in brackets, [ ]. Blanks between a command and its options are optional. Commands may be abbreviated as long as the initial characters uniquely identify the command.

menu
Display a self-test list

Syntax

m[enu]

Function

Display a list describing each EPROM self-test. Figure 15 illustrates the SPU EPROM self-test menu screen.

Example

Figure 15 illustrates the SPU EPROM self-test menu screen.

Figure 15
SPU EPROM self-test menu screen

```
(sfp)> menu
1 - Test 68k instructions used to checksum the EPROM
2 - EPROM checksum verification
3 - Test 68k instructions used to test DRAM
4 - Test 4k DRAM scratch area used for STACK
5 - Test remaining 68k instructions
6 - AMD 9513 interval timer and clock source test
7 - Console UART test
8 - Remote port UART test
9 - Verification of all DRAM except the 1st 64k
A - IMAP pattern and functionality tests
B - PMAP pattern and limited functionality tests
C - Verification of the 1st 64k of DRAM
D - Peripheral interface tests
E - Motorola 68450 DMA controller tests
F - Miscellaneous SPU register tests
(sfp)>
```

preset

Preset nonvolatile soft front panel options

Syntax

p[reset] [s[tandard] | a[lternate-os] | d[iagnostic] | i[nstall]

p[reset] [*eeprom_location* [*data*]]

where

eeprom_location is a hexadecimal number from 0 to 1F.
data is an optional 16-bit hexadecimal number from 0 to 0FFFF.

Function

Set the soft front panel options to a predefined state or view/modify a specific EEPROM location. This command is useful when setting the EEPROM chip initially after a SPU is manufactured or to set all options to a known state. The first form of the preset command requires one argument which specifies the state to set the soft front panel options to. Four predefined states are provided, as described in Table 11.

Table 11
Preset EEPROM states

Soft front panel option	Standard	Alternate	Diagnostic	Install
mode-of-operation	Normal-os	Alternate-os	Diagnostic	Other
boot-device	Disk	Disk	Disk	Tape
location-of-bootstrap	Default	Default	Default	Default
power-up-reboot	Enable	Enable	Disable	Disable
automatic-reboot	Enable	Enable	Disable	Disable
spu-self-test	Enable	Enable	Enable	Enable
test-flags	Normal	Normal	Normal	Normal
remote-port-bps	1200	1200	Unchanged	1200
SCSI-power-up-delay	0xA	0xA	0xA	0xA
user-flags	0x0	0x0	0x0	0x0

The SPU EEPROM is organized into 32 16-bit locations, numbered 0 through 31. These locations are used to store soft front panel options plus other information such as the SPU assembly revision, part number, and so on.

The second form of the `preset` command may be used to view all 32 locations and to set locations 14 through 31. The first argument is a hexadecimal number (0 to `1F`), which specifies the EEPROM chip location. If the second argument is not specified, the contents of the location is displayed. If the second argument is specified, the EEPROM location is modified to the hexadecimal data pattern specified in the second argument.

reset | quit

Restart EPROM as if the **RESET** button had been pressed

Syntax

`r[eset] | q[uit]`

Function

Simulate a reset of the SPU. The 68000 initial program counter (PC) and stack pointer (SP) are fetched and loaded from vectors zero and one, respectively. After the restart, the self-tests execute, if the `spu-self-test` option is enabled. The `test-flags` option affects the self-tests in the standard way.

Note

Soft front panel options affecting command operation:

- `spu-self-test`
- `test-flags`

set

Set a soft front panel option

Syntax

s[et] m[ode-of-operation] = n[ormal-os] | a[lternate-os] | d[iagnostic] | o[ther]

s[et] b[oot-device] = d[isk] | t[ape] | 0 | 3 | 4 | 5 | 6 | 7

s[et] l[ocation-of-bootstrap] = d[efault] | 1[st-copy] | 2[nd-copy] | 3[rd-copy]

s[et] p[ower-up-reboot] = d[isable] | e[nable]

s[et] a[utomatic-reboot] = d[isable] | e[nable]

s[et] s[pu-self-test] = d[isable] | e[nable]

s[et] t[est-flags] = n[ormal] | c[ontinuous] | v[erbose] | a[ll]

s[et] r[emote-port-bps] = 12[00] | 3[00] | 6[00] | 11[0] | 2[400] | 4[800] | 9[600] | 19[200]

s[et] S[CSI-power-up-delay] = hexadecimal value 0 to 0x0FF

s[et] u[ser-flags] = hexadecimal value 0 to 0x0FFFF

Function

Modify soft front panel options. All forms of the set command require the name of the option, followed by an equal (=) sign, followed by the new value for the option.

Most options have a limited set of values that can be assigned. Other options accept a hexadecimal number. The effect of each option is discussed later in this chapter, in the "Soft front panel options" section.

Each time the `set` command is used, the soft front panel is automatically displayed to show the new option state.

Examples

All options and values may be abbreviated to the shortest unique group of characters, as shown in Table 12.

Table 12
Example set options

Command	Meaning
`ss=e`	Enable self-test
`ss=d`	Disable self-test
`sS=0A`	Set SCSI power up delay to 10 seconds
`sS=0`	Disable SCSI power up delay

See the "Soft front panel options" section for a detailed description of each option.

Soft front panel options

Option	mode-of-operation
Function	SPU OS reads this option to determine how the `boot` command behaves. ConvexOS also reads this option to determine how to boot ConvexOS. The option is not used directly by the EPROM. Instead, both the SPU OS `bootstrap` and ConvexOS `boot` scripts read this option. The effects of each value are shown in Table 13.

Table 13
Soft front panel mode of operation

Mode-of-operation	Description
`normal-os`	In this state, when the soft front panel boot command is used to load a SPU OS bootstrap and when the keyswitch is in the **LOCAL** or **REMOTE** position, the bootstrap boots SPU OS from the SCSI device, partition 1 (dk0b), block offset 0, and path name /unix. After SPU OS successfully boots, ConvexOS is booted in multiuser mode.
`alternate-os`	This state is identical to "normal-os" except that the ConvexOS boot script prompts for which mode to boot ConvexOS in (multi or single user).
`diagnostic`	In this state, when the soft front panel boot command is used to load a SPU OS bootstrap and when the keyswitch is in the **LOCAL or REMOTE** position, the bootstrap stops and allows the user to enter the partition number, block offset, and path name. After SPU OS successfully boots, ConvexOS is *not* booted.
`other`	This state is identical to `normal-os` except that after SPU OS successfully boots, ConvexOS is *not* booted.

Option

boot-device

Function

Specifies which SCSI device contains the bootstrap to load when the soft front panel `boot` command is executed. Values are shown in Table 14.

Table 14
Bootstrap devices

Device	Description
Disk	Standard device
Tape	Standard device
0	Foreign device
3	Foreign device
4	Foreign device
5	Foreign device
6	Foreign device
7	Foreign device

Option

location-of-bootstrap

Function

Specifies which copy of the bootstrap to load. For disk devices, it is used to calculate the start block number to begin reading at. The starting disk block (512 bytes per block) for each of the option values is shown in Table 15.

Table 15
Starting disk blocks for bootstrap options

Value	Starting block
Default	1
1st-copy	34
2nd-copy	67
3rd-copy	100

For tape devices, after the tape directory is read and after the bootstrap is found, it continues reading each copy of the bootstrap until the one selected copy is loaded, each copy is 32 512-byte blocks.

Option	power-up-reboot
Function	After a reset from power up condition, this option controls whether or not the EPROM automatically executes the boot procedure without entering the interactive soft front panel (see the `boot` command for boot procedure details). The `automatic-reboot` option must be enabled for this to work. This option is applicable only when the keyswitch is in the SECURE position.
Values	Disabled Enabled

Soft front panel options

Option	automatic-reboot
Function	After ANY reset condition, this option controls whether or not the EPROM automatically executes the boot procedure without entering the interactive soft front panel (see the `boot` command for boot procedure details). This option is applicable only when the keyswitch is in the **SECURE** position.
Values	Disabled Enabled

Option	spu-self-test
Function	If this option is enabled, self-tests 3 through F execute (in addition to 1 and 2) before the interactive soft front panel is entered.
Values	Disabled Enabled

Soft front panel options

Option test-flags

Function Controls verbose message mode during the self-tests and during the `boot` procedure. It also allows the continuous execution of all of the self-tests (or one individual self-test) via the execute command.

Values

The effects of each value are shown in Table 16.

Table 16
Soft front panel mode of operation

Value	Description
`normal`	In this state, both the continuous and verbose modes are disabled.
`continuous`	This state enabled continuous self-test(s) execution. Once the self-tests are started, the only way to terminate execution is by pressing the **RESET** button on the front panel.
`verbose`	In this state, when a self-test is run, a text description of the self-test is output instead of only the identifying hex digit. In addition, when the boot procedure is executed, more information pertaining to the boot is displayed.
`all`	This state is identical to both continuous and verbose modes being enabled at the same time.

Option

remote-port-bps

Function

Specifies the baud rate (bauds per second-bps) to which the remote port is set. It is typically set to 1200 bps.

Values

The baud rate values are shown in Table 17.

Table 17
Baud rate options

Value
110
300
600
1200
2400
4800
9600
19200

Option	SCSI-power-up-delay
Function	Determines how many seconds to wait before the boot procedure is started after a power-up condition. This is necessary for some SCSI devices that require they remain undisturbed while executing their power-up self-tests. The valid delay values are hexadecimal numbers between 0 and FF. To disable this option, clear its value to 0.

Option	user-flags
Function	Not used by the EPROM, but is provided for ConvexOS customization. The valid values are hexadecimal numbers between 0 and FF.

SPU OS operating system

The service processor unit runs SPU OS, an operating system based on UNIX Version 7.

This version of SPU OS comes with many standard UNIX-like utilities, such as `more`, `ls`, and `cat`. These utilities are found in the /bin directory on the service processor, and are described in the *CONVEX SPU OS Utilities Manual.*

SPU OS is a nondemand page version of UNIX, which means that an entire program must be resident in main memory before program execution can begin. If there is not enough memory for a program, the current program in memory will have to be swapped out. This swapping action can have disastrous effects on service processor performance. Because of this, be careful about trying to run many processes at once on the service processor. Specifically, avoid using the ampersand (&) extension for running background processes if possible.

Powering up the system

When the system is powered up, the final state that the machine reaches depends on the setting of the soft front panel flags stored in the EEPROM located on the service processor. If the `automatic-reboot` flag is disabled, the service processor stops at the soft front panel display, where any of the soft front panel commands can be entered.

Booting SPU OS

The service processor can be booted from either the disk or the tape drive. To boot from the tape drive, it is necessary to have a boot image format tape. Booting from tape is desirable only if the disk needs to be reformatted or restored. Booting from tape is never necessary during normal machine operation. When booting from tape, the program that is actually booted is `spu2000`, which is the service processor disk formatter, tester, and file system utility. See Chapter 4, "Service processor peripheral tests (spu2000)".

To boot SPU OS from disk, select the disk as the boot-device, then enter **`boot`**. The `boot` command reads the boot track from the disk and prints a header describing the revision of the boot program.

The program checks the status of the `mode-of-operation` flag in the service processor EEPROM. If the `mode-of-operation` is diagnostic, then the boot program prints a colon and waits for the boot file name to be entered in this format:

`dk` [*major,minor*] *pathname*

Refer to the dk(5) man page. The diagnostic mode of operation is most useful if something besides the default program is needed. In the field, this is necessary only when running spu2000. If the `mode-of- operation` is normal, the colon prompt is never issued and SPU OS is booted on the service processor.

When SPU OS is booted, the first program executed is the Bourne shell script `.profile`. The general sequence of operation by this program is described below:

- Execute `/etc/fsck`, perform a file system check of all mountable file systems.
- Execute `/mnt/bin/.diaginit`, if it exists.
- Execute `/mnt/os/.bootspu`, if it exists.

Using `fsck`

Caution

It is critical to the integrity of the disk files to correct `fsck` errors before attempting to modify the disks in any other way. Failure to do so results in corrupted disk files.

If the service processor runs into a problem while running integrity checks on the service processor file systems, it is mandatory to run the `fsck` file system check program. Conditions that might cause such a problem include a hardware or software failure, or a previous power-down performed incorrectly. A typical message might be as indicated in Figure 16.

Figure 16
Example SPU file system error message

```
SPU mounted file systems check in progress...
/dev/rdk0b: UNALLOCATED I=173 OWNER=root MODE=0
/dev/rdk0b: SIZE=0 MTIME=Jun 25 14:48 1985
/dev/rdk0b: NAME=a.out
/dev/rdk0b: UNEXPECTED INCONSISTENCY; RUN fsck MANUALLY
```

The warning `RUN fsck MANUALLY` means that fsck was unable to determine how best to repair the problems it found on the disk. Consequently, it is necessary to run `fsck` manually. The problem is too difficult for `fsck` to repair without assistance.

To run `fsck` manually, enter:

```
(spu)> /etc/fsck [file_system]
```

where *file_system* is the name of the partition on which the file system is mounted (this name appears to the left of the `UNEXPECTED INCONSISTENCY` message). Except for the root file system, `fsck` should be run on the "raw" (nonbuffered) device of an unmounted file system, for example, /dev/rdk0d. For the root (/) file system, run `fsck` on the "block" (buffered) device, /dev/dk0b. If `fsck` is executed without a *file_system* specified, all file systems in the /etc/fstab file are checked for inconsistencies or corruption.

Once `fsck` is invoked, messages similar to those in Figure 17 can be printed.

Figure 17
Example messages after invoking `fsck`

```
/dev/rdk0b
File System: (unknown)

** Checking /dev/rdk0b
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
UNALLOCATED I=173 OWNER=ROOT MODE=0
SIZE=0 MTIME=Jun 25 14:48 1991
NAME=/a.out
REMOVE ?
```

Entering a **y** in response to the question removes the bad file, in this case, a.out. The file must be removed to restore the file system to a consistent state. If **y** is the response to the query, output similar to Figure 18 can be printed.

Figure 18
Example of removing a file in `fsck`

```
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List
131 files 1170 blocks 808 free
***** FILE SYSTEM WAS MODIFIED *****
***** REBOOT UNIX *****
```

These messages mean that `fsck` was able to restore the disk to a *consistent state*. The data on the disk does not necessarily match the data in SPU OS memory.

If `fsck` modifies the root file system, it may mean that it is necessary to reboot the SPU OS operating system. In this case, reboot with the /etc/reboot program at the `(spu)>` prompt. The `reboot` command returns control to the soft front panel (with the keyswitch in the **LOCAL MAINTENANCE** or **REMOTE MAINTENANCE** positions). Reboot the SPU OS operating system from there as specified earlier.

Often, `fsck` can safely repair problems without operator intervention. In these cases, `fsck` attempts to reboot the SPU OS operating system after completing any file system repairs.

Using `.diaginit`

The `.diaginit` utility is a Bourne shell script that initializes the files needed by the rest of the diagnostic utilities and diagnostic test programs. The main file that must be created is the composite scan ring file, /mnt/usr/lib/scn_ring. This file is a database of all scan rings in the system. It associates logical scan field names with information regarding how to access these fields.

When `.diaginit` is executed, it invokes the `pup` utility to determine the state of the power-up bit in the service processor EEPROM. Whenever this bit is set, `.diaginit` is being executed after a power-up reboot. If the power up bit is not set, then `.diaginit` exits.

If the system has been powered up, the file /mnt/usr/lib/cop.out is moved to /mnt/usr/lib/cop.out.old and the `cop` utility is invoked to create a new cop.out file.

The old and new cop.out files are compared. If they are different, the `scnlink` utility is executed. `scnlink` examines the cop.out file for the ring revision of each board in the system, then it links all ring revision files that correspond to the current system configuration. The state of the power-up bit in the service processor EEPROM is set to `off`. `mminit` initializes the system PCM, which initializes the system configuration files. Last, `scn_util` generates /mnt/boot_db.

Using `.bootspu`

The `.bootspu` script determines if and how ConvexOS should be booted. It invokes the `sfpread` utility to examine the state of the `mode-of-operation` flag contained in the EEPROM on the service processor.

If the `mode-of-operation` flag is set to `normal-os`, then `/mnt/os/boot_cpu` is executed to boot ConvexOS.

If the `mode-of-operation` flag is set to `alternate-os`, a menu of available boot procedures is displayed and the user selects one.

If the `mode-of-operation` flag is set to `diagnostic`, the script exits and a service processor prompt appears.

The system completes the power-up operation in one of the following states:

- At the soft front panel prompt
- At the SPU OS prompt
- At the ConvexOS login prompt

For more information on what can be done at the soft front panel prompt, refer to the *CONVEX System Manager's Guide.*

Diagnostic utilities

There are two kinds of diagnostic utilities:

- Interactive diagnostic utilities
- Error logging utilities

There are two classes of interactive diagnostic utilities:

- Utilities used for hardware initialization of various parts of the system
- Utilities meant for interactive use in manipulating hardware state within the system

Most of the interactive utilities are meant to be run on the service processor from the SPU OS prompt (`spu >`) in an interactive mode.

The error logging utilities are meant to be executed in background on the service processor while ConvexOS is running. The error loggers monitor the various error sources within the CPU.

Diagnostic utilities are generally located in the directory /mnt/bin.

Interactive diagnostic utilities

Several utility programs can be used to support C3400 Series computer processes:

- Cache loading and verifying
- Memory tools
- System initialization
- Hardware state tools

See Table 18 for available service processor diagnostic utilities and a description of their purposes.

Table 18
Interactive diagnostic utilities

Category	Command	Purpose
Control store loaders	`cs`	Loads writable control store(s).
	`commreg`	Examines or modifies the communication registers.
	`dcache`	Dumps the data cache.
	`icache`	Loads, verifies, and dumps the instruction cache.
	`ipte_cache`	Dumps the internal PTE cache.
	`pte_cache`	Dumps the PTE cache.
	`sram`	Dumps the scratch RAM.
Memory tools	`map`	Displays logical-to-physical mapping of main memory.
	`memld`	Loads object file into system memory.
	`mm`	Displays or modifies main memory.
	`mminit`	Initializes main memory.
Clock tools	`jcpu_custom`	Resets the CPU's clock tune values.
	`load_clk`	Loads the CPU's clock tune values.
	`mminit`	Initializes main memory.
	`reset_cpus`	Resets the CPU's clock tune values.
System initialization and setup	`config_chk`	Checks and prints system configuration.
	`diaginit`	Initializes diagnostic description files.
	`initall`	Initializes control stores and main memory.
	`margin`	Sets or reads power supply and system clock margins.
	`mkdiag_db`	Maintains diagnostics configuration file.
	`scn_util`	Hardware initialization utility.
	`scnlink`	Links intermediate scan ring definition file.
	`sysreset`	Resets the computer system.
	`secure`	Enables and disables secure mode.
	`security_clear`	Purges memory and cache.
Hardware utilities	`cop`	Displays board identification information found in COP chips.
	`cpureg`	Initializes or displays the CPU nonvector register state.
	`cpuvreg`	Initializes or displays the CPU vector register state.
	`disable_cpu`	Disables installed CPUs (see `mkdiag_db`).
	`enable_cpu`	Enables installed CPUs (see `mkdiag_db`).
	`errintd`	Error interrupt daemon and logger.
	`hard_logger`	Hard error logger.
	`pup`	Power-up bit read/write utility.
	`syshalt,`	Immediately halts the system on a subsystem basis.
	`vp_scan`	Vector processor scan utility.
	`x`	Hexadecimal/decimal calculator.

Table 18 (continued)
Interactive diagnostic utilities

Category	Command	Purpose
General utilities	`dshell`	System diagnostic test executive.
	`fs`	Field service script utility.
	`iscn`	Interactive scan facility.
	`scn_ring`	Interactive scan ring read/write/check utility.
	`sfpread`	Reads/modifies the SPU soft front panel switches.
	`mm_sniff`	Performs constant main memory error detection.
	`sp2util`	Displays or modifies SPU memory or register locations.

Error logging utilities

While ConvexOS is booted, two diagnostic utilities are being executed on the service processor:

- `errintd`
- `mm_sniff`

These utilities detect and report hardware and environmental errors that occur during the operation of the system.

What `errintd` does

The `errintd` utility is the error interrupt daemon. Its function is to monitor the hardware and environment for errors and to invoke the correct routine to report the errors. It interacts with the `prtlog` utility to print the messages to the console and into the error log file /mnt/errlog.

What `mm_sniff` does

The `mm_sniff` utility periodically reads through main memory looking for single-bit errors. It traverses all physical memory within a programmable time period. When it detects a single-bit error, a scrub cycle is performed to correct the bit. When it comes across a memory address that contains a single-bit error, errintd detects the error and decodes it.

It is important that single-bit errors are detected before they become double-bit errors. A single-bit error can be corrected by the error correcting circuitry on the memory control module (MCM), whereas a double-bit error cannot.

Error monitoring

During operation of the CPU, the service processor monitors for error conditions that might need to be put into the error log. There are three types of errors monitored by the service processor:

- **Environmental errors**—These errors, detected by the system control monitor (SCM), relate to the physical environment of the machine (for example, airflow, power supply, fans, and so on). Depending on how critical the particular problem is, the error reported may be hard or soft.
- **Soft errors**—These errors will not bring the machine to a halt. Often they may be recoverable, as in the case of a soft memory error, where the MCM will correct a single-bit error.
- **Hard errors**—These are fatal errors that will stop the computer immediately. Hard errors may be remedied using the diagnostic tests described in this manual, however, some may be transient.

Each of the three error types has an error logger. Error conditions include: memory soft (single-bit) errors, memory hard (double-bit) errors, microstore parity errors, PTE cache parity errors, bus parity errors, and others. When the service processor detects one of these errors, `errintd` is activated which invokes the applicable error logger and activates the appropriate program to report the error.

After the system is booted, the service processor runs the `prtlog` (print log) program. The /mnt/os/prtlog file monitors all print requests generated by software running on the CCUs or CPUs as well as errintd print requests. Should a board send a message of importance that it wants to write to the console, `prtlog` detects the message, appends it to the error log, and prints it to the console.

When the service processor hardware detects a hard, soft, or environmental error, the service processor signals the sleeping `errintd` (the error interrupt daemon), which manages these error interrupts. It includes the interrupt service functions for soft, hard, and environmental errors.

When single-bit memory errors (soft errors) are detected by the hardware, the service processor enters the device and address (with the help of the error detection and correction code (EDC)) in /mnt/errlog. If a particular RAM device starts having multiple failures, it becomes immediately apparent.

The soft error log file is named /mnt/softlog and is written in printable ASCII characters. The logger decodes the failing address (uninterleaved) and updates the soft error log file. See Figure 19 for a sample soft error log layout.

Figure 19
Example /mnt/softlog display

```
LOG STARTED: Sep 17 15:57 1991
LOG FULL: NO
TOTAL FAILURES:      1079
FAILED DEVICES:         4
ERRORS NOT LOGGED:      0
M                MCM        B S
C                SERIAL     I T
M   DEVICE       NUMBER     T K   FIRST FAIL          LAST FAIL           ADDR    #FAILS
- ------- ------ - - ------------ ------------ ------ ----
LATEST FAILURE:
3e U084A9-U04 01004639 C4    Sep 20 16:58 1991   Oct 23 00:27 1991   0440e8c0   000223
EARLIEST FAILURE:
3e U019F8-U10 01004639 26    Sep 19 09:26 1991   Sep 23 12:50 1991   00bc8cf8   000854
LOGGED FAILURES:
3e U019F8-U10 01004639 26    Sep 19 09:26 1991   Sep 23 12:50 1991   00bc8cf8   000854
3e U084A9-U04 01004639 C4    Sep 20 16:58 1991   Oct 23 00:27 1991   0440e8c0   000223
1e U019A9-U03 01004726 15    Oct 15 13:39 1991   Oct 15 13:39 1991   20b81c58   000001
3e U056F8-U01 01004639  2    Oct 15 13:39 1991   Oct 15 13:39 1991   041560e8   000001
```

When a hard error occurs, the subsystem experiencing the error asserts its hard error line and the processor is halted. `errintd` starts the hard_logger. All `errintd` log messages are written to standard output, piped through `prtlog`, which sends the message to /mnt/errlog and the console. See Figure 20 for a sample hard error log layout.

Figure 20
Example /mnt/errlog display

```
[SPU   @23:54:27] hard_logger: Version:   (Oct 24 11:09:20 CDT 1990)
[SPU   @23:54:28] ME0/MCM: Hard error detected.
[SPU   @23:54:28] ME0/MCM: (HERR500) ECC unit 2: multiple bit error detected:
[SPU   @23:54:28] ME0/MCM: Interleaved address: 00312ad0,   data: 786400e5
[SPU   @23:54:28] ME0/MCM: Actual ECC: 4e, Generated ECC: 41
[SPU   @23:54:29] errintd: Fatal error (status 25).  Cleaning up ...
```

Diagnostic tests

Diagnostic tests are executed under the diagnostic shell (dshell). Diagnostic test programs are contained in the directory /mnt/test.

Test structure

There are two test categories:

- Scan-based tests, such as `pi2_4000` or `mem4100`
- CPU tests, such as `cpu4030` or `cpu4010`

The scan-based tests execute on the service processor. Since scan-based tests only execute test code on the service processor, they do not need the interprocessor communication structure.

The CPU tests execute as separate modules of test code running on both a CPU and the service processor. The CPU tests use an interprocessor communication structure that incorporates main memory and system interrupts to coordinate the separate modules of test code.

The C3400 Series processor diagnostic tests contain a functional test for every subsystem contained in the system.

Diagnostic tests can have two logical groupings:

- Subtests
- Classes

In general, *subtests* test a specific function in the subsystem being tested. A *class* is a group of related subtests. The division between classes and subtests varies greatly between functional tests.

Test strategy

These system tests are functional diagnostics. They indicate if a particular functional unit is operational, but, in general, give no indication of the exact cause of failure (when they are not operational).

Functional tests verify the functional integrity of an entire system or a particular subsystem. Specifically, these tests verify the functionality of the service processor, the CPUs, the main memory, the CPU utility boards and the channel control units. For example, there is a CPU functional test that verifies proper instruction set execution.

Although the cpu functional tests do not isolate failures to a specific board, the main memory functional test isolates a data pattern failure to the failing RAM.

All tests are designed to test from the least complex functions to the most complex functions. Likewise, the tests themselves should be run in an order in which the most basic system functions are tested first.

Table 19 shows the order in which diagnostic tests should be run to verify a system from the ground up, that is, from the most basic function to the entire system.

Table 19
Suggested order of test execution

Program Name	Test Name
`Front panel`	Service processor EPROM-based self-test
`spu2000`	SPU peripheral test (only if unable to boot the SPU)
`spu4000`	Service processor interface test
`SST`	System scan test (see *CONVEX SST User's Manual*)
`mem4100`	Memory subsystem test
`pi2_4000`	PI2 functional test
`cpu4030`	Scalar building block test
`cpu4332`	Enhanced, nonvector, uniprocessor instruction tests
`cpu4331`	C200 Series privileged instructions and architectural features
`cpu4041`	Vector instruction tests
`cpu4241`	Enhanced vector instruction tests
`cpu4333`	Multiprocessor diagnostics

The tests are listed in their suggested order of execution.

Directory structure

The directory structure consists of the root directory (/), and the files that comprise its subdirectories. These files are distributed on tape and are discussed in the "Software distribution tapes" section.

The following list describes what is in each of the subdirectories:

- /hw—Contains interactive scan (`iscn`) scripts.
- /mnt—Contains mountable files that run diagnostic utilities
  - /mnt/bin—Contains the fundamental, frequently used diagnostic utilities.
    * /mnt/bin/CPU—Contains CPU object files for any utility that requires it, such as `mminit`.
  - /mnt/test—Contains all `.t` test programs and processor-specific object codes.
    * /mnt/test/CPU—Contains CPU object codes used by test programs such as `cpu4030.rnn`, `cpu4041.x00`, and `mem4100.x00`.
    * /mnt/test/tables—Contains subtests, classes, and timeout tables for CPU tests.
  - /mnt/os—Contains ConvexOS operating system and utilities, device drivers, EMACS, networking, `vmunix`, and other programs.
  - /mnt/usr—Contains a group of files that are hardware specific.
    * /mnt/usr/scn—Contains `iscn` ring definition files, (such as asp_rev1 and vpd_rev1), the output of the current *cop* configuration scheme (cop.out), and scn_rings, built by `scnlink`, which is the composite of the scan ring definition files.
    * /mnt/usr/lib—This directory contains ASCII databases used by various programs, including DB_cop, and other files.
    * /mnt/usr/ucode—Contains microcode for the vector and scalar processors.
- /tmp—Holds temporary files generated by a variety of programs, such as editors. It is used as scratch pad and is not backed up.
- /bin—Contains an assortment of fundamental, frequently used SPU OS commands, such as `more` and `pwd`.
- /sst—Contains the CONVEX System Scan Test modules.

- /etc—Contains system files and commands, the password file, maintenance files, and all other SPU OS system administration utilities, such as `fsck` and `backup`.
- /stand—Contains the standalone test programs, such as `spu2000`.
- /dev—Contains all special device files, which are those the system uses to route data to particular peripheral devices (such as tape drives or disk drives) or to pseudodevices (such as memory and communication channels).
- /scratch—Contains miscellaneous files that pertain to diagnostic utilities.

Software distribution tapes

The directory structure is built from release tapes that are distributed with CONVEX computers. A brief description of the tapes and their contents are outlined below:

- **System diagnostics**—Contains diagnostic utilities and their object files, test programs and their object files, and various diagnostic timeout tables. The following is a list of the contents of this tape:
  - /mnt/bin
  - /mnt/test
    * /mnt/test/CPU
    * /mnt/test/tables
- **Diagnostic database**—Contains interactive scan (`iscn`) scripts, scan ring definition files, ASCII databases, and microcode for vector scalar processors. The following is a list of the contents of this tape:
  - /hw
  - /mnt/usr
    * /mnt/usr/scn
    * /mnt/usr/lib
    * /mnt/usr/ucode
- **SST**—Contains SST menu system and utilities, and special data files describing the scan rings on each FRU:
    * /sst
- **SPU OS**—Contains SPU OS utilities, special device files, temporary files, system files, system commands, and the kernel. The following is a list of the contents of this tape:
  - /bin
  - /stand
  - /dev
  - /tmp
  - /etc
- **ConvexOS operating system**—This tape contains ConvexOS utilities, device drivers, EMACS, networking, operating system, `vmunix`, and other programs. The following file is on this tape:
  - /mnt/os

For installation procedures, see the release notes for the tape. /mnt/os[mnt/os];directory

Service processor EPROM-based self-tests

3

The service processor EPROM-based self-tests are standalone tests that verify operation of the sections of the service processor unit necessary to boot and execute SPU OS. These tests do not involve any hardware other than the service processor and its associated peripherals.

The self-tests are EPROM-based on the service processor and are not accessible while SPU OS is booted.

There are 15 EPROM- based self-tests, numbered from 1 to F (in hexadecimal). Each individual self-test typically is a group of subtests which target specific hardware.

The hardware tested is divided into three major categories:

- The Motorola 68000 processor and its local memory.
- That part of the service processor logic containing the timer chip and the two universal asynchronous receiver-transmitters (UARTs).
- The boot device interface (for the disk and the cartridge tape), the direct memory access controller (DMAC), and the small computer standard interface (SCSI) and service processor realtime clock registers.

The following hardware is verified functionally:

- The 68000 instruction set
- EPROM integrity via a 32-bit checksum
- The 68000 local DRAM (2 or 8 Mbytes), refresh logic, and memory management logic
- The 68000 peripheral devices:
  - AMD 9513 interval timer
  - 2 each 8251A UARTs
  - Intersil ICM7170 realtime clock/calendar
  - Motorola 68450 4-channel DMA controller
- The SCSI interface (pattern tested only)
- Various CPU and diagnostic control registers (pattern tested only)
- EBUS window map registers (pattern tested only)
- 50/60 hertz line clock interrupt

The following hardware is not verified functionally:

- The diagnostic scan engine connectivity beyond the SPU
- The SPU mass storage devices (disk and tape)
- The realtime clock battery backup
- EBUS window interface to main memory
- The main memory refresh pulse generator
- The SPU copy of the main memory population configuration map (PCM)
- System interrupt bus (transmit and receive)

Prerequisites and required equipment

These self-tests require only a service processor (SP5), a system monitor (JSM) to control power, and a peripheral interface adapter (PI2), and a CPU utilites board (CUJ) to provide clocks.

EPROM-based selt-tests verify the functional areas shown in Table 20.

Table 20
EPROM-based self-tests, functional areas tested

Functional area	Tested by diagnostic	Required—not specifically tested
CPU	No	No
CUJ	No	Yes
Memory even	No	No
Memory odd	No	No
PI2	No	Yes
CCU	No	No
SP5	Yes	No

Test invocation

To run all 15 EPROM-based self-tests (1 through F), the `spu-self-test` soft front panel option must first be set to `enable` via the soft front panel `set` command, prior to invoking the `boot` command.

Default sequence

With the `spu-self-test` option `enabled`, the self-tests automatically execute when the SPU is booted, the **RESET** switch on the front panel is pressed, or the soft front panel `reset` or `quit` commands are executed.

When the `spu-self-test` option is `disabled`, Self-tests 1 and 2 always execute before the soft front panel menu is displayed. Self-tests 3 through F do not execute unless the `spu-self-test` option is `set` to `enable`.

To display the switch settings, enter **`display`** at the `(fp)>` prompt.

Manual sequence

Since the self-tests run before the soft front panel is started, the SPU must then be reset to execute the self-tests. This can be accomplished by pressing the **RESET** switch on the front panel, or by entering **`reset`** or **`quit`** at the `(fp)>` prompt.

SPU LEDs

Before any self-tests are executed, the SPU LEDs' register is pattern-tested with a walking ones and zeroes pattern to ensure correct operation. Self-tests are not executed if a failure is detected. As each self-test executes, the corresponding test number or letter (hexadecimal digit) appears on the screen and is also represented on the four red LEDs on the front edge of the SPU board.

The LEDs are arranged to represent one hexadecimal digit (four binary bits) with the most significant bit <3>on top. Each LED is represented in Table 21.

Table 21
Examples of SPU LEDs

LED bit number	Hexadecimal			
	1	8	9	E
3	OFF	ON	ON	ON
2	OFF	OFF	OFF	ON
1	OFF	OFF	OFF	ON
0	ON	OFF	ON	OFF

Service processor self-test error reporting

In the event defective hardware results in a hung SPU during a self-test, all self-tests are skipped the next time the soft front panel is invoked (after the **RESET** button is pushed or the SPU is powered up). This prevents a deadlock situation since the self-tests automatically run each time the SPU is reset. The following message is output when the above condition has occurred, as shown in Figure 21.

Figure 21
SPU message, skipping self-tests

```
**** WARNING: skipping self-tests ****
A self-test failed to complete when self-tests were last run!
```

To execute all 15 self-tests requires a little over two minutes.

Before each self-test is started, a hexadecimal digit identifying the self-test number is first written to the four SPU LEDs on the front edge of the SPU board. It is then output to the system console. If the `test-flags` option is set to `verbose` or `all`, the description of the self-test is also output following the hexadecimal digit on the system console.

In the event of a self-test error, the last self-test number shown indicates which self-test encountered the error. Typically, an error message describing what went wrong is also output. Many self-tests also output a subtest ID which further isolates where the failure was encountered. The subtest IDs are described in detail under each self-test description.

Any time an error is detected, after all error data is output, the SPU turns on the flashing **ATTENTION** indicator and enters a program loop right after displaying the message, `Halting SPU in tight loop`. At this point you have five options:

RESET	Return to the soft front panel prompt.
+	Restart the soft front panel after clearing all local DRAM.
$	Restart the soft front panel after clearing only the BSS (uninitialized data) area of DRAM used by the EPROM.
!	Restart the soft front panel after no initialization.
*	Simulate a 68000 reset.

To preserve the 68000 stack area (normally beginning at `0x200000`) at the time of an error, the last four options result in the stack being located 8192 bytes below the normal stack area before the soft front panel is restarted. This allows examination of the stack area data at the time of the error upon reentering the soft front panel.

If the `test-flags` option is set to `continuous` or `all`, after Self-test F completes, the self-tests sequence starts over at Self-test 1. This continues until the **RESET** button is pressed.

As stated previously, many self-tests output additional information when a SPU hardware error is detected. Many include a line of the form:

```
test: <id>
```

where `test` identifies the self-test, and `<id>` identifies the specific subtest that failed. In addition, many failures result in a 68000 register dump similar to the one shown in Figure 22.

Figure 22
Example 68000 register dump

```
68k PC: 0000179C, sr: 2704, sp: 00200000
d0-d7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
a0-a7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00200000
```

Subtest descriptions

Table 22 lists the subtest numbers and names and briefly describes each subtest performed.

Table 22
EPROM-based self-test, subtest descriptions

Self-test number	Self-test name	Description
1	1st CPU test	Verifies the 68000 instructions needed to verify the EPROM checksum.
2	EPROM test	Checksums the EPROM.
3	2nd CPU test	Verifies the 68000 instructions needed to verify the DRAM memory.
4	1st DRAM test	Test the 4 kbyte DRAM scratch area used for STACK.
5	3rd CPU test	Verifies the 68000 instructions not previously tested.
6	Timer test	Verifies operation of the timer chip.
7	Console UART test	Verifies operation of the console UART chip.
8	Remote UART test	Verifies operation of the remote UART chip.
9	2nd DRAM test	Performs bit-pattern tests on all DRAM except the 1st 64 kbytes.
A	local DRAM mapper test	IMAP pattern and functionality tests.
B	EBUS main memory mapper test	PMAP pattern and limited functionality tests.
C	3rd DRAM test	Performs bit-pattern tests on the 1st 64 kbytes of DRAM.
D	Peripheral interface tests	SCSI latch test
E	DMAC test	Motorola 68450 DMA controller tests.
F	Register access tests	Miscellaneous SPU register tests.

Self-test 1—1st CPU test

Self-test 1 verifies the minimum 68000 CPU instructions necessary to execute Self-test 2—EPROM test (checksum verification).

Self-test operation

This self-test accesses as little of the EPROM as possible until its contents have been verified.

Error messages

If these instructions fail to verify, Self-test 1 displays:

```
CPU Test Error:  CPU1-<id>
```

where <id> is the failing subtest ID.

Table 23 lists the subtest IDs in Self-test 1.

Table 23
Self-test 1 subtests

Subgroup description	Subtest ID	Instructions verified
Program flow control instructions	10	BEQ <label> BNE <label>

Integer arithmetic instructions	20	CMPB #<data>,Dn
		CMPW #<data>,Dn
		CMPL #<data>,Dn
		LEA <ea>,An
		CMPW #<data>,An
		CMPL #<data>,An
		CMPB <ea>,Dn

Register uniqueness: a0, d0, d1	30	MOVEML <ea>,<A0,D0,D1>
		CMPL <ea>,Dn
		CMPL <ea>,An

Simple 32-bit checksum calculation	40	ADDL <ea>,Dn
		CMPL #<data>,Dn
		DBF Dn,<label>

Self-test 2—EPROM test

Self-test operation

The EPROM is designed with a checksum (using 32-bit addition) equal to zero.

Error messages

If the checksum is not zero, Self-test 2 displays:

```
EPROM Checksum Error.
```

Self-test 3—2nd CPU test

Self-test operation

Self-test 3 verifies the minimum 68000 CPU instructions necessary to execute Self-test 4—1st DRAM test.

Error messages

If this test fails, Self-test 3 displays:

```
CPU Test Error:  CPU2-<id>
```

where <id> is the failing Subtest ID.

Table 24 lists the subtest IDs found in Self-test 3.

Table 24
Self-test 3 subtests

Subgroup description	Subtest ID	Instructions verified
Register verification tests	10	Reg uniqueness test: D0-D7,A0-A7,USP
	11	Reg functionality test: D2-D7,A1-A7,USP

Addressing mode tests	20	MOV An,Dn
		MOV An@,Dn
		MOV An@(offset),Dn
		MOV An@(-offset),Dn
		MOV An@(offset,An:w),Dn
		MOV An@(offset,An:l),Dn
		MOVL An@+,Dn
		MOVW An@+,Dn
		MOVB An@+,Dn
		MOVB An@-,Dn
		MOVW An@-,Dn
		MOVL An@-,Dn
		MOV addr:w,Dn
		MOV addr:l,Dn
		MOV PC@(-offset),Dn
		MOV PC@(offset,An:w),Dn
		MOV PC@(offset,An:l),Dn

Table 24 (continued)
Self-test 3 subtests

Subgroup description	Subtest ID	Instructions verified
Program control instructions test	30	Bcc Label for "cc" of hi,ls,cc,cs,ne,eq,vc, vs,pl,mi,ge,lt,gt,le
Status register access instructions test	40	`MOV <ea>, SR` `MOV SR, <ea>`
Integer arithmetic instructions test	50	`CMPB` condition code generation
	51	`CMPM Ax@+, Ay@+`

	52	`ADDQ #<data>, <ea>`
		`SUBQ #<data>, <ea>`
		`SUB <ea>, An`

Data movement instructions test	60	`MOVEQ #<data>, Dn` `EXG Dx, Dy`
Rotate instructions test	70	`ROL Dx, Dy` `ROL #<data>, Dn`

Bit manipulation instructions test	80	`BSET #<data>, <ea>`
		`BCLR #<data>, <ea>`
		`BCHG Dn, <ea>,`
		`BCHG #<data>, <ea>`

Self-test 4—1st DRAM test

Self-test 4 verifies the DRAM at physical address 0x001FF000 through 0x00200000 (4 k words of stack area).

Self-test operation

This self-test is performed with the 68000 CPU in user mode. The following memory tests are performed:

1. A 16-bit walking ones test, followed by a walking zeroes test, is performed on the first location (0x001FF000) with parity error detection disabled.
2. A true/complement test is performed on the entire range with the 16-bit pattern 0x5555.

   This test first fills every 16-bit location with the ones complement of the previous location, starting with the ones complement of the pattern used. It then fills DRAM again, starting with the uncomplemented pattern. DRAM is then verified to ensure each 16-bit location contains the last written data. The parity error latch is checked at the end of the verify pass.
3. A true/complement test is performed on the entire range with the 16-bit pattern 0xAAAA.
4. A true/complement test is performed on the entire range with the 16-bit pattern 0x5454.
5. A true/complement test is performed on the entire range with the 16-bit pattern 0xA8A8.
6. A 16-bit address uniqueness test is performed on the entire range. The data starts with 0x0001 and increments by 0x0001.
7. The memory parity error detection is verified.

   Parity is not checked in 68000 supervisor mode on the SPU. This is verified by writing inverted parity in supervisor mode and then reading back the data with noninverted parity detection enabled. Parity is then checked in 68000 user mode for both the even and odd bank parity generators by writing inverted parity and reading back with noninverted parity detection enabled. Parity errors are expected in both cases.

Error messages

Up to 10 errors display before this self-test is aborted, allowing you to distinguish between isolated bit failures and gross failures, such as shorts. If less than 10 errors are detected during Self-test 4, the self-test proceeds to its finish, and the SPU is halted at the end of the test.

If the self-test detects a nonparity error type failure, Self-test 4 displays an error message like the one shown in Figure 23.

Figure 23
Self-test 4 nonparity type error message

```
RAM Test Error!
*NOTE* Actual data shown below was re-read after the failure was detected.
Addr: 1FF000, Exp/Act: AAAA5555/00200000
Addr: 1FF004, Exp/Act: AAAA5555/00001130
Addr: 1FF008, Exp/Act: AAAA5555/000031A4
Addr: 1FF00C, Exp/Act: AAAA5555/0000282E
Addr: 1FF010, Exp/Act: AAAA5555/00002842
Addr: 1FF014, Exp/Act: AAAA5555/00002864
Addr: 1FF018, Exp/Act: AAAA5555/00002878
Addr: 1FF01C, Exp/Act: AAAA5555/0000288A
Addr: 1FF020, Exp/Act: AAAA5555/0000289C
Addr: 1FF024, Exp/Act: AAAA5555/000028B2
68k PC: 00353C, sr: 2704, sp 1FFFFA
d0-d7: AAAA5555 FFFFFFFF 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

The error message ends with the 68000 register dump.

If the self-test detects a parity type error, Self-test 4 displays any of the error messages shown in Figure 24.

Figure 24
Self-test 4 parity type error messages

```
68k SUPERVISOR mode RAM Parity Error Interrupt Test Failed!
```

```
68k USER mode EVEN byte Parity Error Interrupt Test Failed!
```

```
68k USER mode ODD byte Parity Error Interrupt Test Failed!
```

Self-test 5—3rd CPU test

Self-test operation

Self-test 5 verifies the remaining 68000 CPU instructions, except `stop`, `#<data>`, and `reset`. It also does some limited exception processing verification including exception stack frame verification for exceptions such as divide by zero, illegal instruction, privilege violation, and address errors.

Error messages

If these instructions fail to verify, Self-test 5 displays:

```
CPU Test Error:  CPU3-<id>
```

where `<id>` is the failing Subtest ID.

Table 25 lists the subtests found in Self-test 5.

Table 25
Self-test 5 subtests

Subgroup description	Subtest ID	Instructions verified

Program flow control instruction tests	10	`BSR <label>`
		`JSR <label>`
		`RTS`
		`RTR`

Set according to condition tests	11	`Scc <ea>`
Data movement instructions tests	20	`LINK An,#<displacement>` `UNLINK An`
	21	`MOVEM <ea>,#<register list>` `MOVEM #<register list>,<ea>`
	22	`MOVEP An(offset),Dn`
	23	`PEA <ea>`
	24	`SWAP Dn`

Table 25 (continued)
Self-test 5 subtests

Subgroup description	Subtest ID	Instructions verified
Integer arithmetic instructions tests	30	ADD condition code generation
	31	SUB condition code generation

	32	CLR <ea>
		ADD <ea>,An
		SUB #<data>,<ea>
		ADD #<data>,<ea>
		ADDQ #<data>,<ea>
		ADDX Dn,Dn
		SUB Dn,<ea>
		SUBQ #<data>,<ea>
		SUBX Dn,Dn
		CMP #<data>,An
		CMP #<data>,<ea>

	33	NEG <ea> NEGX <ea> EXT <ea>
	34	DIVS <ea>,Dn 32 / 16 -> 16r:16q (+/+max16) (+/-max16) (-/-max16) (+1/+max16) overflow check
	35	DIVU <ea>,Dn 32 / 16 -> 16r:16q (max32/max16) (1/max16) overflow check
	36	MULS <ea>,Dn 16 * 16 -> 32 (+max16*+max16) (+max16*-max16) (-max16*-max16)
	37	MULU <ea>,Dn 16 * 16 -> 32 (max16*max16)
	38	TAS <ea>
	39	TST <ea>

Table 25 (continued)
Self-test 5 subtests

Subgroup description	Subtest ID	Instructions verified
Logical instructions tests	40	AND <ea>,Dn OR <ea>,Dn EOR <ea>,Dn NOT <ea>
	41	AND #<data>,<ea> OR #<data>,<ea> EOR #<data>,<ea>
Shift and rotate instructions tests	50	ASL #<data>,Dn ASR #<data>,Dn ROL #<data>,Dn ROR #<data>,Dn LSL #<data>,Dn LSR #<data>,Dn ROXL #<data>,Dn ROXR #<data>,Dn
	51	ASR <ea> ROR <ea> LSR <ea> ROXR <ea>
Bit manipulation operations tests	60	BSET #<data>,<ea> BCLR #<data>,<ea> BCHG Dn,<ea> BCHG #<data>,<ea>
Binary coded decimal instructions tests	70	ABCD Dn,Dn SBCD Dn,Dn NBCD <ea>
System control instructions tests	80	CHK <ea>,Dn RTE TRAP #<vector> TRAPV
Miscellaneous exception condition tests	90	DIVS #0,Dn Divide by zero ADDQB #8,A0 Illegal instruction

Table 25 (continued)
Self-test 5 subtests

Subgroup description	Subtest ID	Instructions verified

User mode privilege violations	91	AND #<data>,SR
		OR #<data>,SR
		EOR #<data>,SR
		MOV #<data>,SR
		MOV An,USP
		RESET
		STOP #<data>

Address errors	92	Long stack frame is verified

Self-test 6—Timer test

Self-test 6 verifies the correct operation of the AMD 9513 interval timer.

Self-test operation

All the chip registers (except the master mode) are pattern tested first with a walking ones and then with a walking zeroes pattern. Next, the following clock sources and outputs are tested:

1. The two output channels of the interval timer are used for BAUD rate generation for the console and remote port UARTs from the 4.9152 MHz clock source.
2. The ac line clock source (50 or 60 Hz) is tested.

Error messages

If the interval timer fails, testing is terminated, and Self-test 6 displays:

```
Timer Error: <ERROR-CODE>
```

where <ERROR-CODE> is the error code identifying the failure.

Table 26 lists the interval timer test error codes.

Table 26
Self-test 6 error codes

ERROR-CODE	Description
1	Pattern test of registers failed.
2	F1 counting came up with differences.
3	F1 source counting tolerance exceeded.
4	Could not clear the line clock interrupt.
5	Failed waiting for next status set.
6	Failed waiting for actual interrupt.
7	Failed to count F1 signal.

Self-test 7—Console UART test

The console 8251 UART test verifies the operation of the system console RS-232 port.

Self-test operation

This self-test verifies only the local port, and gives no indication of the correct operation of the console device itself. First, the RS-232 port is placed in a loopback mode, allowing it to read each character that has been written to the port. All 256 possible byte values are written to and read from the port. Next, each of the interrupts on the port is tested.

Error messages

If the console test detects an error, testing is terminated, and Self-test 7 displays:

```
Console Error: <ERROR-CODE>
```

where <ERROR-CODE> is the error code identifying the failure.

Table 27 lists the console UART test error codes.

Table 27
Self-test 7 error codes

ERROR-CODE	Description
1	Loopback data does not match.
2	Timeout occurred on UART.
3	Framing error from UART.
4	Overrun error from UART.
5	Parity error from UART.
6	Interrupt not received.

Self-test 8—Remote UART test

The remote 8251 UART test verifies the operation of the remote RS-232 port.

Self-test operation

This self-test verifies only the remote port, and gives no indication of the correct operation of device connected to the remote port itself. First, the RS-232 port is placed in a loopback mode, allowing it to read each character that has been written to the port. All 256 possible byte values are written to and read from the port. Next, each of the interrupts on the port is tested.

Error messages

If the remote test detects an error, testing is terminated, and Self-test 8 displays:

```
Remote Error: <ERROR-CODE>
```

where <ERROR-CODE> is the error code identifying the failure.

Table 28 lists the remote UART test error codes.

Table 28
Self-test 8 error codes

ERROR-CODE	Description
1	Loopback data does not match.
2	Timeout occurred on UART.
3	Framing error from UART.
4	Overrun error from UART.
5	Parity error from UART.
6	Interrupt not received.

Self-test 9—2nd DRAM test

Self-test 9 verifies the DRAM beyond physical address 0x00010000 (beyond the 1st 64 kbytes).

Self-test operation

On the SP5, first the lower 4 Mbytes, except the first 64 kbytes, (physical addresses 0x00010000 to 0x00400000) are tested. After all DRAM tests successfully complete for the lower 4 Mbytes, the upper 4 Mbytes (physical address 0x00800000 to 0x00C00000) is tested.

This self-test is performed with the 68000 CPU in user mode. Six subtests are performed for each 4 Mbytes of memory. Each time a subtest is completed, a dot (.) is output to the display to indicate progress, except after the sixth test in each 4 Mbytes. A comma (,) is output between the lower 4 Mbytes and the upper 4 Mbytes. Hence, if all tests pass, Self-test 9 displays:

```
.....,.....
```

The following six memory subtests are performed:

1. A 16-bit address uniqueness test is performed on the entire range. The data starts with a 0x0001 and increments by 0x0001. The data value of zero is not used causing the pattern to shift every 64 kbytes.
2. A true/complement test is performed on the entire range with the 16-bit pattern 0x5555. For a description of the test algorithm see the section on Self-test 4.
3. A true/complement test is performed on the entire range with the 16-bit pattern 0xAAAA.
4. A true/complement test is performed on the entire range with the 16-bit pattern 0x5454.
5. A true/complement test is performed on the entire range with the 16-bit pattern 0xA8A8.
6. A DRAM refresh test, operating in 68000 supervisor mode, is performed on the entire range with the pattern 0xAA5555AA.

   This subtest first fills every 32-bit location with the pattern. It then waits for the rising edge of the next line clock interrupt. Next, it executes a stop instruction (all 68000 bus activity is suspended) and waits for another line clock interrupt. It repeats the stop process 10 times. DRAM is then verified to ensure each 32-bit location still contains the correct data.

Error messages

Up to 10 errors display before this self-test is aborted, allowing you to distinguish between isolated bit failures and gross failures, such as shorts. If less than 10 errors are detected during Self-test 9, the self-test proceeds to its finish, and the SPU is halted at the end of the test.

If any of the subtests detect errors, Self-test 9 displays an error message like the one shown in Figure 25.

Figure 25
Self-test 9 error message

```
RAM Test Error!
*NOTE* Actual data shown below was re-read after the failure was detected.
Addr: 1FF000, Exp/Act: AA5555AA/00200000
Addr: 1FF004, Exp/Act: AA5555AA/00001130
Addr: 1FF008, Exp/Act: AA5555AA/000031A4
Addr: 1FF00C, Exp/Act: AA5555AA/0000282E
Addr: 1FF010, Exp/Act: AA5555AA/00002842
Addr: 1FF014, Exp/Act: AA5555AA/00002864
Addr: 1FF018, Exp/Act: AA5555AA/00002878
Addr: 1FF01C, Exp/Act: AA5555AA/0000288A
Addr: 1FF020, Exp/Act: AA5555AA/0000289C
Addr: 1FF024, Exp/Act: AA5555AA/000028B2
68k PC: 00353C, sr: 2704, sp 1FFFFA
d0-d7: AA5555AA FFFFFFFF 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

Self-test A—Local DRAM mapper tests

Self-test operation

When the DRAM memory mapper is enabled, it translates the logical addresses output by the 68000 CPU to physical DRAM addresses. The lower 4 Mbytes of the 68000 address space are translated by the mapper. The mapper translates a page (4 kbyte block, which begins on a 4 kbyte boundary) at a time. There are 1024 pages of logical address space which can be mapped. The 1024 map registers which control the mapper are implemented in 2 kbytes of static RAM at SPU addresses `0x00FF8000` through `0x00FFA000`.

Self-test A, the local DRAM mapper pattern and functionality test, verifies the map register static RAM by performing the following four RAM subtests on the map registers:

1. A 16-bit walking ones test, followed by a walking zeroes test is performed on the first location (`0x00FF8000`) with parity error detection disabled.
2. A 16-bit address uniqueness test is performed on the entire range. The data starts with `0x0001` and increments by `0x0001`.
3. A true complement test is performed on the entire range with the 16-bit pattern `0x5555`.

   For a description of the test algorithm see the section on Self-test 4.
4. A true complement test is performed on the entire range with the 16-bit pattern `0xAAAA`.

   For a description of the test algorithm see the section on Self-test 4.

Error messages

Up to 10 errors display before this self-test is aborted, allowing you to distinguish between isolated bit failures and gross failures, such as shorts. If less than 10 errors are detected during Self-test A, the self-test proceeds to its finish, and the SPU is halted at the end of the test.

If any of the subtests detect an error, Self-test A displays an error message like the one shown in Figure 26.

Figure 26
Self-test A general error message

```
RAM Test Error!
*NOTE* Actual data shown below was re-read after the failure was detected.
Addr: FF8000, Exp/Act: AAAA5555/00200000
Addr: FF8004, Exp/Act: AAAA5555/00001130
Addr: FF8008, Exp/Act: AAAA5555/000031A4
Addr: FF800C, Exp/Act: AAAA5555/0000282E
Addr: FF8010, Exp/Act: AAAA5555/00002842
Addr: FF8014, Exp/Act: AAAA5555/00002864
Addr: FF8018, Exp/Act: AAAA5555/00002878
Addr: FF801C, Exp/Act: AAAA5555/0000288A
Addr: FF8020, Exp/Act: AAAA5555/0000289C
Addr: FF8024, Exp/Act: AAAA5555/000028B2
68k PC: 00353C, sr: 2704, sp 1FFFFA
d0-d7: AAAA5555 FFFFFFFF 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

After pattern testing the map registers, several functional tests are performed on the mapper. Each 16-bit map register consists of a 12-bit page frame and the following four access control bits:

- Valid
- Read
- Write
- Execute

When the 68000 CPU is in user mode, the mapper self-test checks all bits to validate memory accesses. When it is in supervisor mode, only the valid bit is checked. The remaining subtests verify the address translation, as well as the behavior of the access control bits. Subtests 1 and 2 verify address translation.

Since turning on the mapper disables the EPROM, some EPROM code must be copied to DRAM and the map registers must be setup before the mapper is activated. When enabled, the page offset portion of the address (address bits <0> through <11>) is passed through unaltered. The mapper simply replaces bits <12> through <23>.

Since the executable mapper test code is fetched with the mapper on, a test scheme is used which results in correct execution, even if the mapper is broken. The first two subtests copy the code thread which tests the mapper to the same page offset in every physical page. Therefore, even if the mapper is broken (the wrong bits <12> through <23> are generated), the code still executes.

If an error is detected, the mapper is disabled and the code thread jumps back into the EPROM. This scheme requires the code thread, interrupt vectors (1 kbyte), and stack space must fit into 4 kbytes. Each page contains:

- Interrupt vectors (256)
- Executable test code that verifies the mapper
- Executable interrupt handling code to handle unexpected exceptions
- A tag word which uniquely identifies that physical page
- Stack space at the end of the page

The EPROM code sets everything up before turning over control to the RAM based code thread which enables the mapper.

If either of the first two subtests detects an error, Self-test A displays an error message like the one shown in Figure 27.

Figure 27
Self-test A mapper error message

```
IMAP Test Error:  <id>
Logical-Addr: 000000, Mapreg-Addr: FF8000
   Physical-Addr: Exp/Act: 000000/001000
Logical-Addr: 002000, Mapreg-Addr: FF8004
   Physical-Addr: Exp/Act: 002000/001000
  .
  .
  .
68k PC: 002A54, sr: 2704, sp 1FFFFA
d0-d7: 00000400 00000001 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

In Figure 27, <id> is the ID of the failing subtest.

If errors are detected by the first two subtests, but the error count is less than 10, the tests proceed to their finish, and the SPU is halted at the end of the tests. This allows you to distinguish between isolated bit failures and gross failures such as shorts.

In the event of an error in the remaining subtests (`0x10` through `0x85`), the error message is identical to the previous example, except only the subtest ID and the 68000 registers are displayed.

Table 29 lists all of the local DRAM mapper subtests.

Table 29
Self-test A subtest IDs

Subgroup description	Subtest ID	Description
Address translation verification (supervisor mode)	1	Addresses `0x00000000` to `0x00400000`
	2	Addresses `0x00800000` to `0x00C00000`
Valid logical address to invalid physical address verification	10	Read access via logical address `0x00200000`
	11	Write access via logical address `0x00200000`
Verification of mapper valid bit in 68000 supervisor mode	20	Read access to invalid page
	21	Write access to invalid page
	23	Read access to valid only page with no R/W/E access
	24	Write access to valid only page with no R/W/E access
	25	Execute access to valid only page with no R/W/E access
Verification of 68000 user mode I/O space access (physical address `0x00C00000` to `0x00FFFFFF`) via user I/O bit in the mapper control register (bit <1> in `0xFFD018`)	40	Read access with user I/O disabled
	41	Write access with user I/O disabled
	50	Read access with user I/O enabled
	51	Write access with user I/O enabled

Table 29 (continued)
Self-test A subtest IDs

Subgroup description	Subtest ID	Description
Verification of access control bits in 68000 user mode	60	Read access to invalid page with R/W/E access
	61	Write access to invalid page with R/W/E access
	62	Execute access to invalid page with R/W/E access
	70	Read access to valid page with R access (no W/E)
	71	Write access to valid page with W access (no R/E)
	72	Execute access to valid page with E access (no R/W)
	80	Read access to valid page with W/E access (no R)
	81	Write access to valid page with R/E access (no W)
	82	Execute access to valid page with R/W access (no E)
Verification of mapper fault detection	83	Verification that access to a map register with bad parity via a logical to physical address translation produces the correct 68000 bus error in the bus error source register (BSR) at `0xFFD020`
	84	Verification that access to a map register with bad parity via a 68000 read access produces the correct bus error in the BSR
	85	Verification that a byte access to a map register via a 68000 read access produces the correct bus error in the BSR

Self-test B—EBUS main memory mapper tests

This self-test verifies the EBUS window map registers, excluding actual accesses to main memory via the EBUS.

Self-test operation

First the map register static RAM is verified by performing the following four RAM tests on the map registers:

1. A 16-bit walking ones test, followed by a walking zeroes test, is performed on the first location (`0x00FF8000`), with parity error detection disabled.
2. A 16-bit address uniqueness test is performed on the entire range. The data starts with a `0x0001` and increments by `0x0001`.
3. A true complement test is performed on the entire range with the 16-bit pattern `0x5555`.

   For a description of the test algorithm see the section on Self-test 4.
4. A true complement test is performed on the entire range with the 16-bit pattern `0xAAAA`.

   For a description of the test algorithm see the section on Self-test 4.

Error messages

Up to 10 errors display before this self-test is aborted, allowing you to distinguish between isolated bit failures and gross failures, such as shorts. If less than 10 errors are detected during Self-test B, the self-test proceeds to its finish, and the SPU is halted at the end of the test.

If any of the subtests detects an error, Self-test B displays an error message like the one shown in Figure 28.

Figure 28
Self-test B general error message

```
RAM Test Error!
*NOTE* Actual data shown below was re-read after the failure was detected.
Addr: FFA000, Exp/Act: AAAA5555/00200000
Addr: FFA004, Exp/Act: AAAA5555/00001130
Addr: FFA008, Exp/Act: AAAA5555/000031A4
Addr: FFA00C, Exp/Act: AAAA5555/0000282E
Addr: FFA010, Exp/Act: AAAA5555/00002842
Addr: FFA014, Exp/Act: AAAA5555/00002864
Addr: FFA018, Exp/Act: AAAA5555/00002878
Addr: FFA01C, Exp/Act: AAAA5555/0000288A
Addr: FFA020, Exp/Act: AAAA5555/0000289C
Addr: FFA024, Exp/Act: AAAA5555/000028B2
68k PC: 00353C, sr: 2704, sp 1FFFFA
d0-d7: AAAA5555 FFFFFFFF 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

After pattern testing the map registers, limited functional tests are performed on the mapper. The valid bit functionality and the map register access is verified. Operations requiring access to main memory are not tested.

If the functional tests detect an error, testing is terminated, and Self-test B displays an error message such as shown in Figure 29.

Figure 29
Self-test B functional test error message

```
PMAP Test Error:  <id>
68k PC: 00353C, sr: 2704, sp 1FFFFA
d0-d7: AAAA5555 FFFFFFFF 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

In Figure 29, <id> is the ID of the failing subtest.

Table 30 lists all of the mapper functionality tests and a description of each.

Table 30
Self-test B subtest IDs

Subgroup description	Subtest ID	Description
Verification of map register valid bit in 68000 Supervisor mode	10	Read access to invalid page (BSR is verified)
	11	Write access to invalid page (BSR is verified)
Verification of mapper fault detection	40	Verification that access to a map register with bad parity as a result of a window access at `0x00800000` produces the correct 68000 bus error in BSR
	41	Verification that access to a map register with bad parity via a 68000 read access produces the correct bus error in the BSR
	42	Verification that a byte access to a map register via a 68000 read access produces the correct bus error in the BSR

Self-test C—3rd DRAM test

Self-test C verifies the DRAM at physical address 0x00000000 through 0x00010000 (the 1st 64 kbytes of DRAM).

Self-test operation

This test is performed with the 68000 CPU in user mode. When the EPROM is enabled it occupies the lower 64 kbytes of address space. When the EPROM is disabled, DRAM occupies the lower 64 kbytes of address space.

In order to test the low 64 kbytes of address space, the EPROM is copied to physical address 0x00010000 and set up as logical address 0x0000000. The local DRAM memory mapper is then enabled. The following memory tests are performed:

1. A true complement test is performed on the entire range with the 16-bit pattern 0x5555.

   For a description of the test algorithm see the section on Self-test 4.
2. A true complement test is performed on the entire range with the 16-bit pattern 0xAAAA.
3. A true complement test is performed on the entire range with the 16-bit pattern 0x5454.
4. A true complement test is performed on the entire range with the 16-bit pattern 0xA8A8.
5. A 16-bit address uniqueness test is performed on the entire range. The data starts with a 0x0001 and increments by 0x0001. The data value of zero is not used causing the pattern to shift every 64 kbytes.
6. A DRAM refresh test (operates in 68000 supervisor mode) is performed on the entire range with the pattern 0xAA5555AA.

   For a description of the test algorithm see the section on Self-test 9.

Error messages

Up to 10 errors display before this self-test is aborted, allowing you to distinguish between isolated bit failures and gross failures, such as shorts. If less than 10 errors are detected during Self-test C, the self-test proceeds to its finish, and the SPU is halted at the end of the test.

If any of the subtests detects an error, Self-test C displays an error message like the one shown in Figure 30.

Figure 30
Self-test C error message

```
RAM Test Error!
*NOTE* Actual data shown below was re-read after the failure was detected.
Addr: 000000, Exp/Act: AAAA5555/00200000
Addr: 000004, Exp/Act: AAAA5555/00001130
Addr: 000008, Exp/Act: AAAA5555/000031A4
Addr: 00000C, Exp/Act: AAAA5555/0000282E
Addr: 000010, Exp/Act: AAAA5555/00002842
Addr: 000014, Exp/Act: AAAA5555/00002864
Addr: 000018, Exp/Act: AAAA5555/00002878
Addr: 00001C, Exp/Act: AAAA5555/0000288A
Addr: 000020, Exp/Act: AAAA5555/0000289C
Addr: 000024, Exp/Act: AAAA5555/000028B2
68k PC: 00353C, sr: 2704, sp 1FFFFA
d0-d7: AAAA5555 FFFFFFFF 0007FFFB 00010000 00003CA0 0000FF00 000026CA 0F0F0F00
a0-a7: 001FF028 00200000 001FF000 0000000A 00003526 0000340A 0000190C 001FFFFA
```

Self-test D—Peripheral interface tests

This self-test pattern tests the SCSI interface latch used to communicate with the service processor disk and tape devices.

Self-test operation

While the pattern test is performed, SCSI reset is held asserted. Upon completion of the pattern test, SCSI reset is no longer held asserted.

If any of the subtests detects an error, Self-test D displays an error message like those shown in Figure 31.

Figure 31
Self-test D error messages

```
SCSI Error: Reset assert failed Exp/Act: 04/00
```

```
SCSI Error: Pattern test Exp/Act: 0A/00
```

```
SCSI Error: Reset clear failed Exp/Act: 00/04
```

```
SCSI Error: Select assert failed Exp/Act: 02/00
```

The expected and actual values (`Exp/Act`) in the above messages can vary depending on the point of failure.

Self-test E—DMAC tests

The Motorola 68450 direct memory access controller (DMAC) test performs memory-to-memory transfers to check the integrity of the DMAC. DMAC testing occurs as follows:

1. A pattern test is performed on the DMAC's general control register (GCR).
2. The test cycles through the four DMAC channels; each channel is fully tested before the test continues with the next channel.
   a. The registers associated with that channel (shown in the following list) are pattern tested.
      * DCR—Device control register
      * OCR—Operation control register
      * SCR—Sequence control register
      * MTCR—Memory transfer count register
      * MAR—Memory address register
      * DAR—Device address register
      * BTCR—Base transfer count register
      * BAR—Base address register
      * NIVR—Normal interrupt vector register
      * EIVR—Error interrupt vector register
      * MFCR—Memory function code register
      * CPR—Channel priority register
      * DFCR—Device function code register
      * BFCR—Base function code register
   b. The test performs interrupt terminated memory-to-memory transfers to ensure that the DMAC moves data correctly.

If any of the subtests detects an error, Self-test E displays an error message like the one shown in Figure 32.

Figure 32
Self-test E general error message

```
DMAC Register Pattern Test Failed:
Register: <WWWWWWWW>  Address: <XXXXXXXX>
Pattern: <YY> (<ZZZZ>)  Mask: <AA>
Actual: <BB>  Expected: <CC>
```

In Figure 32:

<WWWWWWWW>	Register name
<XXXXXXXX>	Register address
<YY>	Pattern value
<ZZZZ>	Pattern name
<AA>	Register mask (bits that exist in the failing register)
<BB>	Actual value read from the register
<CC>	Expected value to be read from the register

Rebound is a failed pattern test.

Next, memory-to-memory moves are performed and interrupts used to indicate completion. Should an error occur when performing a move or checking a transfer for accuracy, Self-test E displays an error message like the one shown in Figure 33.

Figure 33
Self-test E transfer error message

```
<string>

channel:  <X>  base address: <YYYYYYYY>
<register>  <initial value>  <current value>
<register>  <initial value>  <current value>
<register>  <initial value>  <current value>
.
.
.
```

In Figure 33:

<X>	Channel number
<YYYYYYYY>	Channel address
<string>	One of the messages in Figure 34

Figure 34
Self-test E transfer error message substrings

```
Timeout while waiting for DMAC interrupt
```

```
Received error interrupt from DMAC
```

```
Received wrong interrupt from DMAC
```

```
DMAC move buffer not as expected
```

Self-test F—Register access tests

The miscellaneous tests check the values contained in the SCSI control register and the real time clock registers, to ensure they contain reasonable values. Should a register be out of range, Self-test F displays an error message like the one shown in Figure 35.

Figure 35
Self-test F error message

```
Register: <WWWWWWWW>  Address: <XXXXXXXX>
Actual: <AAAA>  Minimum: <YYYY>   Maximum: <ZZZZ>
```

In Figure 35:

<WWWWWWWW> Register name

<XXXXXXXX> Register address

<AAAA> Actual value read

<YYYY> Minimum allowable value

<ZZZZ> Maximum allowable value

Table 31 lists the miscellaneous registers, their address, and a minimum and maximum value for each.

Table 31
Miscellaneous registers

Register	Address	Min. value	Max. value
SCSI control register	0xffd00c	0	0
RTC 1/100 second	0xffd040	0	99
RTC hours	0xffd042	0	23
RTC minutes	0xffd044	0	59
RTC seconds	0xffd046	0	59
RTC month	0xffd048	0	12
RTC day	0xffd049	0	31
RTC year	0xffd04c	0	99
RTC day of week	0xffd04e	0	6

Hardware debugger

From the soft front panel, you can enter the hardware debugger with the `debug` command. The debugger, in turn, has its own set of commands. The hardware debugger prompt is `(dbg)>`.

Functions included in the hardware debugger are:

- A symbolic 68000/68010 disassembler
- A command to print the b.out symbol table for the entire EPROM
- Memory `peek/poke` via symbolic or absolute hexadecimal addresses
- Command to switch SPU processor between user and supervisor modes
- A command to enable/disable the local DRAM memory mapper
- The ability to set up SPU processor registers and call a subroutine
- A command to translate a symbolic address to absolute and vice versa
- A mode for calibration of the realtime clock
- A command to copy the bootstrap program from tape to disk, or vice versa
- A command which provides tape functions (skip file mark and rewind)

Table 32 lists and describes the commands available in the hardware debugger. The commands are described in greater detail in the following pages.

Table 32
Hardware debugger commands

Command	Description
a	Translate symbolic address to absolute address or vice versa.
c	Enter realtime clock/calendar chip calibration mode.
d	Disassemble from specified symbolic or absolute address.
j	Call 68000 subroutine at specified address via `jsr`.
md	Disable SPU memory mapper.
me	Copy EPROM to physical address `0x10000`.
mb	Modify or dump local memory in bytes.
ml	Modify or dump local memory in longwords.
mw	Modify or dump local memory in words.
p	Print EPROM symbol table.
r	Display or set 68000 registers for use with the `j` command.
Sd	Raw copy from tape device to disk bootstrap.
Sr	Reset SCSI interface and print interface state.
Ss	Display current SCSI interface state.
St	Copy SCSI disk bootstrap to tape device.
Tr	Issue SCSI rewind command to tape device.
Ts	Issue SCSI skip filemarks command to tape device.

Within the debugger all addresses may be entered in symbolic or absolute hexadecimal form. Symbolic addresses may also contain a offset from the symbol, such as

```
spu2000+6
```

Upon entering the debugger, the state of the local DRAM memory mapper is reported along with the 68000 mode (user or supervisor).

a

Translate between symbolic address and absolute address

Syntax

a *hex-address* | *symbolic-address*

Function

Translates a 68000 PC address from a register dump or logic analyzer trace to its symbolic address form or translates a symbolic address to the absolute hexadecimal address. When entering a hexadecimal address, be sure the leading character is a numeric character (0 through 9).

Example

An example address translation is shown in Figure 36.

Figure 36
Example address translation function

```
(dbg)> a spu2000+6
spu2000+0x6: 0x00001136
(dbg)> a 113a
0x0000113A: spu2000+0xA
(dbg)>
```

C
Enter realtime clock/calendar chip calibration mode

Syntax

c

Function

Places the realtime clock (ICM7170) in a mode where it pulses pin 12 every second when in perfect calibration. The clock source can then be adjusted until the pulses occur every second on pin 12. When calibration is complete, the user can return the realtime clock to normal operation by pressing **RETURN**.

Example

An example realtime clock is shown in Figure 37.

Figure 37
Example realtime clock function

```
(dbg)> c
Real time clock calibration enabled.
RTC Pin 12 - 1 Second Interrupts.
Enter carriage return when finished.
(dbg)>
```

d

Disassemble from specified symbolic or absolute address

Syntax

`d [ -d | -h ]` *hex-address* | *symbolic-address*

Function

Provides a symbolic disassembler similar to `adb`. The register offsets and or immediate value may be displayed in hexadecimal or decimal via the `-h` or `-d` option, respectively. Hexadecimal is the default display mode. The starting address may be either a hexadecimal address or a symbolic address. Each line listed contains:

1. The absolute hex address of the instruction
2. The symbolic address enclosed in brackets (or hex if 2048 bytes away from the nearest symbol)
3. The instruction mnemonic and its operands

Initially 16 lines of disassembled instructions display, at which time the output pauses. Then you have three options:

- Type **q** to quit and terminate the disassembly.
- Press **RETURN** to disassemble one more line and pause.
- Press the **SPACEBAR** to disassemble 16 more lines and pause.

Example

An example of disassembled code is shown in Figure 38.

Figure 38
Example disassemble code function

```
(dbg)> d spu2000
0x1130 [spu2000]: clrl    d5
0x1132 [spu2000+0x2]:   movl    d5,IER:w
0x1136 [spu2000+0x6]:   movw    d5,MCR:w
0x113A [spu2000+0xA]:   movw    #0x0032,d3
0x113E [spu2000+0xE]:   dbf     d3,spu2000+0xE
0x1142 [spu2000+0x12]:  movb    d5,TN+0x1:w
0x1146 [spu2000+0x16]:  clrl    d1
0x1148 [spu2000+0x18]:  clrw    CPR:w
0x114C [spu2000+0x1C]:  movl    d1,d2
0x114E [spu2000+0x1E]:  moveq   #0x0F,d6
0x1150 [spu2000+0x20]:  lea     spu2000+0x28:w,a5
0x1154 [spu2000+0x24]:  jmp     coprd:w
. . .
(dbg)>
```

j

Call 68000 subroutine at specified address via `jsr`

Syntax

`j` *hex-address* | *symbolic-address*

Function

Allows a subroutine to be called via the 68000 `jsr` instruction convention. The 68000 register copy set up with the `r` command is loaded and the 68000 jumps to the specified address using `jsr`. If the called subroutine returns, the resultant 68000 registers are saved and displayed.

Example

The following example calls a register dump function. After return from the function, the registers are dumped again, as illustrated in Figure 39, and saved as noted previously.

Figure 39
Example 68000 subroutine call function

```
(dbg)> j cregdmp
68k PC: 000074B2, sr: 2704, sp: 00110000
d0-d7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
a0-a7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00110000
sr: 00002000
d0-d7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
a0-a7: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00110000
(dbg)>
```

md
Disable SPU memory mapper

Syntax

```
md
```

Function

Forces the 68000 to execute from the EPROM. Disables the local DRAM memory mapper which results in the EPROM being selected and accessed in the lower 64 kbytes of 68000 address space. In addition, all 68000 address are not translated by the mapper (for example, 68000 address is equal to physical address). After the mapper has been disabled, the state of the mapper is reported along with the 68000 mode (user or supervisor).

Example

An example of disabling the SPU memory mapper is shown in Figure 40.

Figure 40
Example disable SPU memory mapper function

```
(dbg)> md
Mapper is disabled.  68k supervisor mode.
(dbg)>
```

me

Copy EPROM to physical address 0x10000, set up and enable memory mapper

Syntax

```
me
```

Function

Forces the 68000 to execute from a copy of the EPROM in DRAM. Upon execution of this command, the mapper is first disabled. Next the EPROM is copied to physical address `0x10000` (just beyond the EPROM) and checksum verified.

If the copy is successful, the mapper is set up so that logical address (that is, 68000 address) `0x0` is mapped to physical address `0x10000`. The mapper is then enabled. This allows the user to `peek` and `poke` DRAM with the mapper enabled. This is useful due to the fact that when SPU OS is booted, the mapper is alway enabled. The mapper behaves differently based on the 68000 mode (user or supervisor). The `u` or `s` command can be used in conjunction with this command to attempt to emulate "normal operation."

After the mapper has been disabled, the state of the mapper is reported along with the 68000 mode (user or supervisor).

Example

An example of setting up and enabling the SPU memory mapper is shown in Figure 41.

Figure 41
Example set up and enable memory mapper function

```
(dbg)> me
Mapper is enabled.  68k supervisor mode.
(dbg)>
```

mb

Modify or dump local memory in bytes

mw

Modify or dump local memory in words

ml

Modify or dump local memory in longwords

Syntax

[mb | mw | ml] *begin-address*

[mb | mw | ml] *begin-address end-address*

[mb | mw | ml] *begin-address,count*

Where

begin-address	Starting address (symbolic or absolute hexadecimal)
end-address	Ending address (applies to the second command form only)
count	Number of elements to display (applies to the first command form only)

Function

In the form mb *begin-address*, this command enters an interactive mode which allows modification of memory.

In the form mb *begin-address end-address*, all elements from *begin-address* up to and including *end-address* are displayed.

In the form mb *begin-address,count*, *count* elements from the starting address are displayed.

The following list gives all of the valid responses while in interactive mode:

Response	Description
RETURN	Advance to next address
hex-value	Write optional value to current address, advance to next address
hex-value=	Write optional value to current address, and stay at the present address
—	Back up to the previous address
q	Exit interactive mode.

Example

An example of displaying memory is shown in Figure 40.

Figure 42
Example modify or dump memory function

```
(dbg)> mw IER
<FFD010> 1 0=
<FFD010> 0 q
(dbg)>
```

The first line of the above sequence modifies the IER at address `0xFFD010` to 0, reread and display the new value.

The second line quits interactive mode.

p
Print EPROM symbol table

Syntax

```
p
```

Function

Displays the entire symbol table, one page at a time. Two columns of symbol and address are output per line. The output is paused every 16 lines or 32 symbols. There are two options:

RETURN or **SPACEBAR**	List 16 more lines and pause.
`q`	Quit and terminate the symbol display.

EXAMPLE

An example symbol table display is shown in Figure 43.

Figure 43
Example symbol table display function

```
(dbg)> p
ABTEXC:          0x00000404          ihard:          0x00001576
DMPREGS:         0x000005B6          ienv:           0x0000157C
INVINT:          0x00000662          ITIMER:         0x00001582
Excid:           0x0000112C          ipwr:           0x00001588
spu2000:         0x00001130          ABORT:          0x000015CA
get_upctype:     0x000012DE          RESET:          0x00001670
exsfp:           0x000013AC          HEX:            0x00001686
clr_bss:         0x000013B8          PRTMSG:         0x000016DE
selft_run:       0x000013DA          PRTCHR:         0x00001722
SELF_T:          0x000013E8          CPUERR:         0x00001760
STESTrtn:        0x0000144E          st_tab:         0x000017A0
ispur:           0x0000152E          stst_cnt:       0x000017DC
trap1:           0x00001534          HEXTB:          0x00001A30
trap2:           0x00001544          REGTB1:         0x00001A40
isoft:           0x0000156A          REGTB2:         0x00001A80
ipbus:           0x00001570          REGTB3:         0x00001A84
   . . .
(dbg)>
```

r

Display or set 68000 registers for use with the j command.

Syntax

`r` [*REG* = *value* [*REG* = *value* [,*value* ...]]

Where

REG is one of the following:

d0,d1,d2,d3,d4,d5,d6,d7,
a0,a1,a2,a3,a4,a5,a6,a7,
sr

value is a symbolic or absolute hexadecimal value.

Function

Sets up or displays a copy of the 68000 registers for use with the `j` command. The actual 68000 registers are not affected, only the memory copy. Initially, upon entering the hardware debugger, the stack pointer (A7) and the status register (SR) are set to reasonable values. The stack pointer is set to `0x100000`. Normally, this is out of the way of the typical stack which starts at `0x200000` and grows downward. The `SR` is copied when the debugger is entered.

If no arguments are specified the current copy of the registers is displayed. When one or more registers are specified, the unspecified registers are not modified. When specifying multiple register names (as in the forth coming example), a blank must separate each name/assignment specification.

Each register may be assigned an absolute hexadecimal value or a symbolic address. If desired, one invocation of the command can set all registers by following each value with a comma and more values. When commas are used, the progression is from `d0` through `a7` and then `sr`.

After each invocation of the command, the new copy of the registers are displayed.

Examples

The command in Figure 44 sets `d0` and `a5` only:

Figure 44
Example set up or display registers function

```
(dbg)> r d0=25 a5=spu2000
sr: 000027 04
d0-d7: 00000025 00000000 00000000 00000000 00000000 00000000 00000000 00000000
a0-a7: 00000000 00000000 00000000 00000000 00000000 00001130 00000000 00100000
(dbg)>
```

The command in Figure 45 sets `d0` through `d3`, `a5` through `a6`, and `sr`.

Figure 45
Example set up or display registers function

```
(dbg)> r d0=10,20,30,40 a5=50,_edata sr=2000
sr: 00002000
d0-d7: 00000010 00000020 00000030 00000040 00000000 00000000 00000000 00000000
a0-a7: 00000000 00000000 00000000 00000000 00000000 00000050 0000ef04 00110000
(dbg)>
```

Sd

Raw copy from tape device to disk bootstrap

Syntax

`Sd` [*block-count*]

Function

Copies *block-count* 512-byte blocks from the current tape position to the disk boot location currently specified by the SFP option `location-of-bootstrap`. If no *block-count* is specified a *block-count* of 32 is used.

Sr

Reset SCSI interface and print interface state

Syntax

```
Sr
```

Function

Asserts the SCSI interface reset for a few microseconds. The state (SCSI phase) of the SCSI interface is printed after the reset is complete.

Ss
Display current SCSI interface state

Syntax

```
Ss
```

Function

Prints the state (SCSI phase) of the SCSI interface.

St

Copy SCSI Disk Bootstrap to tape device

Syntax

`St` [*block-count*]

Function

Copies *block-count* 512-byte blocks from the disk boot location currently specified by the SFP option `location-of-bootstrap` to the current tape position. If no *block-count* is specified a *block-count* of 32 is used.

Tr

Issue SCSI rewind command to tape device

Syntax

```
Tr
```

Function

Issues the SCSI rewind command to the tape device and waits for it to complete. Initially the command waits for the target to be ready before the command is executed.

Ts

Issue SCSI skip filemarks command to tape device

Syntax

`Tr` [*count*]

Function

Issues the SCSI skip filemarks command to the tape device and waits for it to complete. Initially the command waits for the target to be ready before the command is executed. If the *count* is omitted, a *count* of one is assumed.

68000 exceptions

The service processor 68000 CPU can be interrupted by a variety of internal exceptions and/or external interrupts. Various self-tests induce exceptions or interrupts in a controlled fashion.

If an exception or interrupt occurs at a point where it was not expected, the exception is reported and the 68000 registers are displayed. For bus and address error exceptions, the access address, 68000 function codes, and the bus error source register are all displayed. The SPU is halted.

Figure 46 illustrates several unexpected exceptions.

Figure 46
Example exception messages

```
Unexpected Interrupt/Exception: 68k bus error: 02
Access address: 800000, fcode: 1 (user data), Read Access
Error source (BSR): 10<68k-dtack-timeout>
68k PC: 006C34, sr: 0704, sp: 1FFE3C
d0-d7: 00000001 00000000 00000000 00000005 00000002 00000001 00000000 00000000
a0-a7: 00800000 FFFFD01B 00000000 00000000 0000137E 001FFE69 001FFF2C 001FEE2E
```

```
Unexpected Interrupt/Exception: 68k bus error:  02
Access address: 000000, fcode: 1 (user data), Write Access
Error source (BSR): 01<IMAP-err>
68k PC: 006D9A, sr: 0700, sp: 1FDE3C
d0-d7: 00000001 00000002 00000000 00000000 00000002 00000001 00000000 00000035
a0-a7: 00000000 FFFFD01B 00000000 00000000 0000166C 001FDE61 001FDF2C 001FCE2E
```

```
Unexpected Interrupt/Exception: 68k address error:  03
Access address: 000001, fcode: 5 (supervisor data), Read Access
Error source (BSR): 01<IMAP-err>
68k PC: 006D9A, sr: 0700, sp: 1FDE3C
d0-d7: 00000001 00000002 00000000 00000000 00000002 00000001 00000000 00000035
a0-a7: 00000000 FFFFD01B 00000000 00000000 0000166C 001FDE61 001FDF2C 001FCE2E
```

```
Unexpected Interrupt/Exception: 68k illegal instruction:  04
68k PC: 006D9A, sr: 0700, sp: 1FDE3C
d0-d7: 00000001 00000002 00000000 00000000 00000002 00000001 00000000 00000035
a0-a7: 00000000 FFFFD01B 00000000 00000000 0000166C 001FDE61 001FDF2C 001FCE2E
```

Table 33 lists the exceptions and interrupts on the service processor.

Table 33
Service processor exceptions and interrupts

Vector	Source	Description
0x02	68000	68k bus error
0x03	68000	68k address error
0x04	68000	68k illegal instruction
0x05	68000	Divide by zero
0x06	68000	68k chk instruction trap
0x07	68000	68k trapv
0x08	68000	68k user mode privilege violation
0x09	68000	68k trace trap
0x0A	68000	Line 1010 emulator
0x0B	68000	Line 1111 emulator
0x18	68000	Spurious interrupt
0x19	68000	Level 1 autovector
0x1A	68000	Level 2 autovector
0x1B	68000	Level 3 autovector
0x1C	68000	Level 4 autovector
0x1D	68000	Level 5 autovector
0x1E	68000	Level 6 autovector
0x1F	68000	Level 7 autovector
0x20	68000	Trap #0x0
0x21	68000	Trap #0x1
0x22	68000	Trap #0x2
0x23	68000	Trap #0x3
0x24	68000	Trap #0x4
0x25	68000	Trap #0x5

Table 33 (continued)
Service processor exceptions and interrupts

Vector	Source	Description
0x26	68000	Trap #0x6
0x27	68000	Trap #0x7
0x28	68000	Trap #0x8
0x29	68000	Trap #0x9
0x2A	68000	Trap #0xA
0x2B	68000	Trap #0xB
0x2C	68000	Trap #0xC
0x2D	68000	Trap #0xD
0x2E	68000	Trap #0xE
0x2F	68000	Trap #0xF
0x5F	External	SCSI interface
0x7E	External	68450 (DMA) error interrupt
0x7F	External	68450 (DMA) normal completion interrupt
0x9C	External	Console UART transmit ready
0x9D	External	Remote UART transmit ready
0x9E	External	Console uart receive ready
0x9F	External	Remote uart receive ready
0xB6	External	System soft error
0xB7	External	System interrupt BUS interrupt
0xDC	External	System hard error
0xDE	External	50/60 Hz line clock interrupt
0xDF	External	9513 interval timer/channel 5
0xFE	External	Local memory parity error
0xFF	External	Power up/power down interrupt

EPROM initialization

When the service processor is reset, either by a power up or by pressing the **RESET** button, the 68000 is also reset. Table 34 lists the various hardware initialization steps taken by the EPROM after reset.

Table 34
EPROM hardware initialization steps after reset

Step	Description
1	68000 CPU is reset.
2	All external interrupts disabled.
3 [1]	Local DRAM memory mapper is disabled.
4	Delay approximately 65 µs (to allow SCM power up interrupt).
5 [1]	Clear SPU LEDs (power interrupts restart execution at step 6).
6 [1]	Clear front panel **ATTENTION** and **RUN** LEDs.
7	Read remote port baud rate from EEPROM.
8	Initial baud rate generator for console and remote (AMD9513).
9	Read remote port baud rate from EEPROM.
10	Initialize console UART.
11	Initialize remote port UART.
12	Enable/initialize realtime clock/calendar.
13	Initialize main CPU control registers (turn off clocks, etc.).
14	Set main memory interleave to 8-way.
15	Initialize main memory refresh period register.
16	Initialize system interrupt bus receive mask for interrupt `0xB`.
17	Initialize all 1024 EBUS window map registers to zero.
18	Check if power up/down interrupt has occurred since reset:
19	—If power up interrupt expected, continue from step 23.
20	—If power up interrupt unexpected, print error message and halt SPU.
21	—If power down interrupt, print error message and halt SPU.
22	—No power interrupt, continue at step 25.
23	Set `reset-from-power-up` flag in EEPROM.

[1] Indicates that the specified hardware should have already been initialized if an actual hardware reset occurred.

Table 34 (continued)
EPROM hardware initialization steps after reset

Step	Description
24	Acknowledge power up interrupt.
25	Check environmental monitor register for errors, print message if errors present.
26	Identify microprocessor type (68000 or 68010); set front panel **RUN** LED if 68010.
27	Clock the bus error source register by generating a bus error.
28	Read EEPROM and check if the previous self-test pass completed. If not, reset `selftest-in-progress` flag, and continue at step 30.
29	Run self-tests (1 and 2 always; 3 through F if enabled).
30	Copy EPROM to physical DRAM, beginning address `0x10000`.
31	Map EPROM copy to logical address `0x0` and enable DRAM mapper.
32	Enter soft front panel, or process automatic boot if keyswitch is in the **SECURE** position.

Service processor peripheral tests (spu2000)

4

Test program `spu2000` is a dual-purpose, standalone test/utility program that tests the cartridge tape and disk interfaces. It performs three functions:

- Tests and formats tape or disk
- Overwrites the data area of the disk
- Restores the SPU OS root partition

The test/utility function formats and tests the disk and cartridge tapes. The wipe function overwrites the data area of the disk. The SPU OS root restore function uses tapes created by /etc/backup to create a UNIX root on the Winchester or IOmega disk.

The `dshell` utility does not support this standalone test/utility program.

User interface

This program does not operate under the dshell.

Boot from tape

One way to execute this test program is to boot a `/etc/backup` format tape. The `backup` script copies the `spu2000` program to the boot location on the cartridge tape. Then, the test program boots and the first prompt displays.

Boot from disk

A faster method of executing this test program can be used if the root directory exists on the Winchester disk. The soft front panel `mode-of-operation` must be set to diagnostic.

Procedure

The boot procedure is almost the same from tape or disk.

1. Change the system switch to **LOCAL**.
2. If SPU OS is running, enter:

   ```
   /etc/reboot
   ```

3. The soft front panel selection menu is displayed on the screen.
4. If SPU UNIX is NOT running, press the **RESET** button to display the front panel menu. At the `(fp)>` prompt, enter:

   ```
   boot
   ```

5. The following message is displayed (disk only):

   ```
   SPU OS boot (Generated:. . .)
   ```

6. At the `(fp)>` prompt (disk only), enter:

   ```
   dk(1,0)stand/spu2000
   ```

The program will execute, starting with the main menu, as described in the next section.

SPU disk/tape utility main menu

Whether `spu2000` is executed from tape or disk, the first-level prompt presents four options, as illustrated in Figure 47.

Figure 47
SPU disk/tape diagnostic utility main menu

```
SPU Disk/Tape Diagnostic Utility $Revision: 1.3 $

      (U) for UNIX Root Restore
      (D) for Disk/Tape Utility
      (S) for SPU Hardware Utility
      (R) for Reboot SPU

Enter utility to execute -
```

These options are discussed in the following sections.

SPU OS root restore

When the UNIX root restore function is selected in the disk/tape utility main menu with a **u**, the program reads the date of the backup and displays the SPU OS root partition restore display, as illustrated in Figure 48.

Figure 48
SPU OS root partition restore display

```
UNIX Root Partition Restore
    reading bad block table...
      Attempting sector: 0 Successful.
    reading root date code ...
      Attempting sector: 0 Successful.

SPU UNIX root size = 2052 blocks. Backed up Mon Aug 19 21:48:39 1991
```

After the back-up date displays, the program asks whether to restore to a (Winchester) disk or an IOmega disk (with the valid selections in brackets and the default in parentheses), as shown in Figure 49.

Figure 49
SPU OS root partition restore query

```
Restore root onto disk or IOmega? [di] (d)
Define the device to recover
```

After you enter a disk type, the program asks you to enter input disk parameters, or use the defaults, for the selected disk display (refer to the "Format/test function" subsection in the next section).

Finally, the screen prompts you for permission to continue (with the valid selections in brackets and the default in parentheses), as shown in Figure 50. Either 0 or 1 can be selected for the recovery copy.

Figure 50
SPU OS root partition restore confirmation

```
Recover the root at this time? [yn] (n)
Recover copy 0 or 1? [01] (0)
```

When the SPU OS root partition recovery is complete, you are returned to the main menu.

SPU disk/tape utility menu

When the disk/tape utility function is selected in the disk/tape utility main menu with a `D`, the program displays the SPU disk/tape utility menu as illustrated in Figure 51.

Figure 51
SPU disk/tape utility menu

```
SPU Disk/Tape Utility

      (D) for Disk (SPU Winchester)
      (T) for Tape (SPU cartridge)
      (I) for IOmega (SPU removable disk)
      (E) for Exit Test

Enter controller type/function -
```

Caution — **Be careful when entering responses, as the format utility is data destructive.**

Standard defaults

If you enter `D`, `T`, or `I`, the program asks the question in Figure 52 (with the valid selections in brackets and the default in parentheses).

Figure 52
SPU disk/tape utility format query

```
Format desired using standard defaults and no prompts [yn] (y)?
```

If you enter `y`, you will not be permitted to test, only to format the device. You are asked to confirm your answer, as shown in Figure 53.

Figure 53
SPU disk/tape utility confirmation

```
ALL PREVIOUS DATA WILL BE DESTROYED. ARE YOU SURE [yn] (n)?
```

Repeats

If you enter **y** to the format query and the confirmation, the device is formatted without any further operator response. When the format finishes, the program asks whether it should do another format, as shown in Figure 54.

Figure 54
SPU disk/tape utility format repeat query

```
Format another with defaults and no prompts [yn] (n)?
```

If you enter **y**, the automatic format operation repeats. If you enter **n**, you are returned to the disk/tape utility main menu.

Tape parameters

If you answered **T** to the first prompt (for tape), the optimum parameters are displayed, but cannot be changed.

Disk parameters

If you answered **D** or **I** to the first prompt (for Winchester or IOmega disk), and you answered **n** to the standard defaults, you can do more than format the device.

You are first prompted with disk information. The program displays the disk parameter menu for the type of disk you are using.

The menu for a Winchester disk (with the defaults shown in parentheses), is illustrated in Figure 55.

Figure 55
SPU Winchester disk parameter menu

```
SPU Winchester Disk Parameters:

    Number of heads           (6)     ->
    Number of cylinders       (320)   ->
    Start of write precomp    (128)   ->
    Step rate                 (2)     ->
    Sector data size          (512)   ->
    Sectors per track/head    (18)    ->
    Logical drive number      (0)     ->
    Sector interleave         (3)     ->

Are all inputs correct? - [yn] -> y
```

If you answer **n** to the prompt, you are allowed to change any of the parameters.

The optimum parameters, listed in Table 35, have been initialized in the program as the defaults.

Table 35
SPU disk/tape format defaults

Parameter	Tape	Winchester	IOmega
Number of heads	6	6	1
Number of cylinders	251	320	306
Start of write precomp	NA	128	0
Step rate	NA	2	0
Sector data size	1024	512	512
Sectors per track/head	17	18	64
Logical drive number	0	0	0
Sector interleave	1	3	16

Caution

Although it is possible to change these parameters from the optimum values, it is not recommended.

Format/test function

After displaying the parameters, the program displays the prompt in Figure 56 (with the valid selections shown in brackets). The program returns to this prompt after every test.

Figure 56
SPU disk/tape utility format/test query

```
Format/test, Debug, or Abort operation [F,D,A]?
```

If you enter **A**, the `Abort operation` option returns you to the SPU disk/tape utility menu.

If you enter **F**, the `Format/test` option causes the program to continue with the subtest enable menu, which is illustrated in the next subsection.

Note

At this time, D, the `Debug` option, is nonfunctional and returns the user to the disk/tape utility menu.

Subtest enable

Each subtest can be enabled separately. You can select options such as loop on test and the maximum number of errors to be allowed. The test prompts display one at a time until they are all on the screen, as illustrated in Figure 57.

Figure 57
SPU format/test utility subtest prompts

```
   Run maintenance track test?     [yn]    (n) ->
   Run format test?                [yn]    (n) ->
   Run WIPE disk?                  [yn]    (n) ->
   Run WIPE verify                 [yn]    (n) ->
       start sector                  (      0) ->
       end sector                    (284479) ->
       num sectors               [1-128] (128) ->
       number of passes           [1-10] (  1) ->
       pass 1 pattern                (random) ->
       pass 2 pattern                (random) ->
   Run write test?                 [yn]    (n) ->
   Run read test?                  [yn]    (n) ->
   Run bad block fix?              [yn]    (n) ->
   Run random read test?           [yn]    (n) ->
   Run seek test?                  [yn]    (n) ->
   LOOP ON TESTS?                  [yn]    (n) ->
   MAX NUMBER OF ERRORS?                   (1) ->
Are all inputs correct?             [yn]        ->
```

Caution

Run these tests only in the order shown. A write test must be selected before a read test can be attempted because the read test performs a *data compare* using the pattern the write test generates.

Subtest descriptions

This section describes the `spu2000` subtests in execution order, according to the peripheral test prompts as illustrated in Figure 57.

Maintenance track subtest

This subtest applies to IOmega disks only. It displays the data stored on the maintenance track and allows changes in some areas. The data display consists of five parts. An example of the maintenance track subtest display is shown in Figure 58.

Figure 58
Maintenance track subtest data display

```
                Running Maintenance track subtest

                   MAINTENANCE TRACK DATA

Bad Track Log:
      bad   replacement
      track track
      ----  ----
      No bad tracks have been flagged

Other Data:
      Idle time before auto stop of spindle (minutes) ---> 5
      Write verify ------------------------------------------> y
      Check ECC ---------------------------------------------> y
      Interleave --------------------------------------------> 8

Change maintenance track data [yn] (n)? n
```

Bad track log
Indicates all tracks flagged as bad, along with the number of the alternate track being used.

Auto stop of spindle
Specifies the amount of idle time before stopping the spindle. The time displayed is the time the drive remains spinning while not in use. The purpose of stopping the spindle is to prevent excessive wear of the media under the read/write head.

Write verify flag
Provides for automatic read of any written sector and performance of a CRC check when the flag is y. Setting of the flag causes a slowdown of the I/O since each time a track is written, a second revolution of the disk is required to verify the written blocks. This technique, however, provides the advantage of detecting a write error at write time, which allows a retry of the write operation.

ECC checking flag
Provides for automatic calculation of an ECC for the entire track whenever one or more sectors on a track are written, if the flag is y. Each time data is written to a track, the entire track must be read to recompute the ECC, and then the ECC block must be written. This can require as many as three revolutions of the disk. Having the flag set at y provides the capability to correct unrecoverable read errors. The ECC can recover an entire 256-byte block which has become unreadable.

Interleave value
Indicates the number to be added to the last sector number. This number indicates the next consecutive sector to be used for a read or write operation. Interleaving sectors allow processing time between reads or writes. If set properly, the next desired sector should be coming under the read/write heads very shortly after processing of the last sector is completed.

Any of the maintenance data except the bad track data can be changed. If the ECC flag is changed to y, or the interleave value is changed, a format subtest automatically runs (whether specified or not). This is necessary to make the disk usable. If ECC is already set to y, the format subtest does not execute (unless enabled).

Figure 59 shows an example of the input screen for the maintenance track subtest.

Figure 59
Changing maintenance track data

```
Change maintenance track data [yn] (n)? y
      Idle time before auto stop of spindle (minutes)
            [5/7.5/10/.../30/d (for disabled)]        (5)    ->
      Write verify
            [y/n]                                     (y)    ->
      Check ECC (Disk will be reformatted if change to 'y')
            [y/n]                                     (n)    ->
      Interleave (Disk will be reformatted if change)
            [1/2/4/8/16/32]                           (8)    16


WARNING - THE DISK WILL BE REFORMATTED!

Are all inputs correct [yn]? y

      maintenance track change underway ...

      Maintenance track subtest ----> passed          0:00:25
      Running Format subtest ------------> passed     0:01:30
```

Format subtest

This subtest formats Winchester, IOmega, or cartridge tape media. The subtest uses the format parameter defaults or the parameters selected from the maintenance track subtest, with one exception. If an IOmega is being formatted and the interleave value was changed in the previous maintenance track subtest, then that interleave value is used during formatting.

When an IOmega is being formatted and either a `NO TRACK 0` or `NO INDEX SIGNAL` error occurs, the message shown in Figure 60 is displayed.

Figure 60
Maintenance track disk cartridge load prompt

```
Reinsert disk cartridge. Press 'return' when ready
```

Reinserting the disk and pressing **RETURN** forces the IOmega controller to reread the maintenance tracks in hopes that another attempt will succeed. Rebuilding the maintenance tracks may allow the disk to be usable again.

If one of these two errors still occurs on the second read, the message and prompt shown in Figure 61 is displayed.

Figure 61
Maintenance track disk cartridge error message

```
Error-maintenance tracks on IOmega cartridge may be bad. Rebuilding
these tracks results in loss of the current bad track history.
Rebuild maintenance tracks [yn] (n)
```

WIPE disk subtest

The WIPE disk subtest allows the user to rapidly write over the entire data area of the SPU disk.

Parameters

The subtest allows you to specify the following:

- Number of WIPE passes (1-10). The default is 1 pass.
- Data pattern to use on each pass. You can specify either a 4-byte pattern, a multi-sectored block of random data, or continuously random data. The default is block random data.

Table 36 illustrates typical user choices.

Table 36
WIPE disk options

Option	Result
0	Causes a 4-byte pattern of `0x00000000` to be replicated across the disk.
F	Causes a 4-byte pattern of `0xFFFFFFFF` to be replicated across the disk.
123	Causes a 4-byte pattern of `0x23123123` to be replicated across the disk.
abcd	Causes a 4-byte pattern of `0xabcdabcd` to be replicated across the disk.
random 1	Causes a multi-sectored block of random data to be replicated across the disk. The number following the word *random* is used as a seed for the random number generator, and is optional. If no seed is specified, the SPU's clock tick is used. The seed is displayed at runtime, so WIPE can be repeated with the same data.
RANDOM 99	Causes continuous random data to be replicated across the disk. *Runtime is 5-6 times longer than for a fixed pattern, or for the block random data.* The number following the word *RANDOM* is used as a seed for the random number generator, and is optional. If no seed is specified, the SPU's clock tick is used. The seed is displayed at runtime, so WIPE can be repeated with the same data.

- Sector area to WIPE. You can specify the start and end sectors. The default is sector 0 though the last sector (the whole disk).
- WIPE verification. Each WIPE pass can be read verified. This option can be used as a fast verify of read/write capability over the entire disk. Use of this option more than doubles the run time. The default is NO verify.
- Number of sectors. You can specify how many sectors (1-128) are written to the disk at a time (per SCSI I/O operation). This number also corresponds to the size of the multi-sectored block used for generating block random data. The smaller this number, the longer the run time due to increased disk I/O inefficiency. The default is 128.

Caution

This subtest destroys existing data on the SPU disk. You can restore data only if you have first run /etc/backup to a good backup tape.

The parameters are displayed and you are prompted for possible changes before the subtest begins.

WIPE display

While the WIPE program is running, it displays information as illustrated in Figure 62.

Figure 62
WIPE display

```
Running WIPE disk pass 1 pattern=random  seed=7127  0:29:53
Running WIPE disk pass 2 pattern=00000000x   0:29:52

Completed   1 loop(s), passed   1,  0 Error(s) detected  0:59:45
```

Write and read subtests

The write subtest repeats a fixed pattern, `0xE5A55A5E`, the number of times necessary to fill a block. This is written to every block which the drive parameters indicate exist. For instance, if only 100 tracks are specified on a 306 track IOmega, then only the first 100 tracks are written.

The read subtest reads each block which the drive parameters indicate exists and compares each block with the fixed pattern, `0xE5A55A5E`. The test keeps all blocks that generate errors in an internal table. These blocks are marked as bad during the bad block fix subtest so they will no longer be used.

Bad block fix subtest

When this subtest begins, you are prompted to enter manually the number of each bad block (if any) that was not previously flagged by the read subtest as being bad. Then the test sorts the blocks into ascending order and writes them to the disk/tape. After the blocks are logged on the disk/tape, they are no longer available for data storage. Instead, alternate blocks are used.

Random read subtest

This subtest performs 100 reads to randomly calculated block numbers. Before the test begins, the test determines the maximum number of blocks on the disk. Block numbers are calculated in the range zero to the maximum minus one.

Seek subtest

This subtest performs an accordion seek in reverse. A seek is made from `min` to `max` track, then to `min+1`, `max-1`, `min+2`, `max-2`, and so forth. The accordion seek ends with a one-track seek. Each time the head comes to rest on a track, the test reads the middle block of the track to verify that the seek was successful.

Other subtest options

The final two options in Figure 57 allow you to specify looping on subtests and the maximum number of errors that can occur before testing is aborted. If you choose to loop, the subtests loop indefinitely, unless you choose one of the following options:

- Press `A`, which returns you immediately to the main menu. Since this action may leave the drive in an unknown state, the next option should be used.
- Press either `B` or `C`, which terminates the test in a controlled manner. Thus, a few seconds may pass before this termination actually occurs. If a format was in process, the subtest must complete before termination occurs.
- The test reaches the maximum number of errors which causes test termination.

Execution times

Execution times for format/test subtests, as illustrated in Table 37, are maximum under nominal conditions.

Table 37
Test execution times

Subtest	Number of minutes		
	Tape	Disk	IOmega
Maintenance track	NA	NA	1.0
Format	10.0	1.5	1.5
Write	10.0	9.5	17.0
Read	20.0	11.0	10.0
Bad block fix	0.5	1.5	1.5
Random read	NA	0.5	0.2
Seek	NA	1.5	0.8
Total	40.5	25.5	32.0

Run times per pass, as illustrated in Table 38, were benchmarked using an 18 Mbyte (formatted) drive on a C1, and a 154 Mbyte (formatted) Micropolis 1375 disk drive on a C220.

Table 38
WIPE disk runtimes
(minutes per pass)

Data pattern	Read verify	C1 18Mb	C220 154Mb
Fixed	no	2	13
Blocked random	no	2	13
Full random	no	10	70
Fixed	yes	4	30
Blocked random	yes	4	30
Full random	yes	20	156

Service processor hardware utility

When D, the service processor hardware utility function, is selected in the SPU disk/tape utility main menu, the program invokes this standalone utility. The service processor hardware utility functions the same as the utility `sp2util` (refer to Chapter 5, "Diagnostic utilities"), but with reduced capability due to its standalone operation.

Error messages

The `spu2000` program reports I/O errors in a standard format. However, if additional data is available at the time of the error, that data is also reported. The basic format for the errors is illustrated in Figure 63.

Figure 63
`spu2000` error message format

```
spu2000: <error desc> on <device> drive <dev.#> during command <cmd>.
Operation: <operation> [optional data displays here and on next line]
```

In Figure 63:

`error desc`	The type of error that occurred, such as, `NO INDEX SIGNAL, DRIVE NOT READY, SEEK ERROR.`
`device`	One of the following: `Adaptec, IOmega,` or `Tape.`
`dev.#`	The device number. Each device begins at 0 with additional drives of the same type being numbered 1, 2, and so on.
`command`	The I/O command to the drive which resulted in the error; for example, `READ SECTOR, WRITE SECTOR, REZERO UNIT, SEEK.`
`operation type`	The name of the subtest, such as, `Format, Read, Bad Block Fix, Disk Restore, Seek.`
`optional data`	The block being accessed if using a disk (Adaptec or IOmega) and a read, write, or verify is underway.

The stream, segment, and sector is printed, if using a tape error is detected by the controller, or a data compare error was detected by the service processor.

A third line is sometimes printed, which contains the position of the error within the block (or sector on tape), the expected data, and the actual invalid data when using either a tape or disk and a data compare error occurs.

Examples of error messages that can occur are illustrated in Figure 64.

Figure 64
spu2000 error message examples

```
spu2000: NO INDEX SIGNAL on IOmega drive 0 during command REZERO UNIT Operation.
Format
```

```
spu2000: ID CRC ERROR on tape drive 0 during command READ SECTOR.
Operation. Read.    Stream. .    Segment. 14.    Sector. 14
```

```
spu2000: DATA COMPARE ERROR on Adaptec drive 1 during command READ SECTOR.
Operation. Read. .    Block. 4523
Bytes into block. 2. Expected. 0xE5A55A5.  Actual: 0x00000000
```

```
spu2000: ID CRC ERROR &
RECORD NOT FOUND on tape drive 0 during command WRITE SECTOR.
Operation. Write.   Stream. .   Segment. 20.   Sector. 5
```

Notice that the last error message reports more than one error. The tape controller sets a series of bits to indicate which errors have occurred. The error message for each error prints.

Error descriptions

This section provides an alphabetical list of errors generated by `spu2000`, with a brief description of the error.

`BAD ARGUMENT`

The Winchester disk drive detected a bad value in one of the fields of a command block.

`BAD SEEK`

The Winchester disk drive was unable to find the requested track.

`COMMAND TIMEOUT`

The tape drive has taken an excessively long time performing a head load, seek, or head positioning.

`DATA ADDR MARK NOT FOUND`

Each time a sector is written on any disk drive, a fixed pattern is written just before the sector; this is called the data address mark. If this pattern is not detected when an attempt is made to read the sector back, the drive has no idea where the data begins. Thus, the drive reports a data address mark not found.

`DATA COMPARE ERROR`

Data just read from any peripheral drive does not match with an expected data pattern. The data in error is printed.

`DATA CRC ERROR`

For any peripheral device, the cyclic redundancy check (CRC) algorithm combines all the bytes in a sector and generates a 2-byte field which is written after the data. Upon read back, the bytes being read are again combined, using the same algorithm, and the result is compared with the 2 bytes at the end of the data. If the bytes do not match, this error occurs.

DATA XFER NOT COMPLETE

The sector buffer in the IOmega drive either was not completely written to the disk or was not completely transferred to the service processor when reading from the disk.

DMA TIMEOUT

Transfer of data to or from the IOmega's internal memory by the IOmega's direct memory access controller failed.

DRIVE ALREADY BUSY

The service processor is about to perform an I/O operation to the tape but finds that the tape controller is busy doing an unrequested operation.

DRIVE NOT READY

The peripheral drive is not ready to perform any I/O operations. This error generally occurs with the cartridge tape, although it can occur from any of the peripherals. Once a tape has been inserted, it takes a maximum of 3 minutes for the tape to reposition itself. The error message displays if a test was started on the tape several seconds before the tape was inserted, and the tape did not go ready before the timeout.

ECC ERROR DURING VERIFY

The Winchester drive is unable to correct an error using the error correction code.

FIFO WOULD NOT EMPTY

The first in, first out (FIFO) data buffer on the tape controller was not empty at the completion of an I/O operation.

ID ADDR MARK NOT FOUND

The disk drive writes a fixed pattern at the beginning of each ID (header) called the ID address mark. Each time the controller reads or writes a sector, it checks this address mark for validity first. Since the controller automatically performs retries, multiple bad reads of an ID address mark have already occurred before this error is reported.

ID CRC ERROR

For any peripheral device, the checksum calculated during the read of an ID does not match the cyclic redundancy check sum that follows the ID. This error only prints when all multiple retries to read the ID fail.

ILLEGAL BLOCK ADDRESS

The disk drive was sent a block (sector) address outside the valid range for the peripheral.

ILLEGAL INTERLEAVE

An interleave factor other than 1, 2, 4, 8, 16, or 32 was requested for the IOmega. For the Winchester, an interleave less than one or greater than 17 was requested.

INSUFFICIENT CAPACITY

No room remains to store data; the IOmega is full.

INVALID BLOCK NUMBER

The tape controller detected an attempt to perform I/O to an invalid block on the cartridge tape. Valid block numbers must be 0 to 25601.

INVALID COMMAND

The disk drive controller received an unrecognizable I/O command.

INVALID LOGICAL UNIT NO.

An attempt was made to access a Winchester drive which does not exist.

INVALID STREAM NUMBER

For tape, an attempt was made to write to a stream number other than 1-6.

LOST DATA

The host sent data too rapidly to the tape controller causing a loss of some of the data.

MEDIA NOT LOADED

This error occurs when the IOmega cartridge is not loaded with the door securely closed and an I/O attempt is made.

NO INDEX SIGNAL

The disk drive medium is not spinning, or it is not spinning at the correct speed.

NO SEEK COMPLETE

The disk drive detects a bad seek.

NO TRACK 0

On the Winchester, this message indicates a seek to track zero failed. For the IOmega, it means the maintenance track data (all 8 copies) is bad.

RECORD NOT FOUND

For any peripheral device, the ID field for the requested block could not be found. Generally, this means I/O errors are preventing the ID from being read.

UNCORRECTABLE DATA ERROR

For any disk drive, an error in the data field of a block was detected. Either the ECC correction was not enabled or the error was too large for ECC correction to recreate the data.

UNFORMATTED OR BAD FORMAT

The Winchester drive controller detected a flawed format and aborted the current I/O operation.

VOLUME OVERFLOW

The Winchester disk is full; there is no room to store additional data.

WRITE FAULT

For any disk drive, an internal drive error resulted in an inability to complete the requested write operation. This error will occur if the automatic write verify is turned on and the verify fails.

WRITE PROTECTED

An attempt was made to write to an IOmega or tape cartridge that is write protected.

Diagnostic utilities 5

This chapter describes publicly accessible diagnostic utilities in alphabetic order.

The two classes of diagnostic utilities documented in this chapter are:

- Utilities used for initialization of various parts of the system hardware
- Utilities used for manipulating the hardware state within the system

Most of these utilities are meant to be run from the SPU OS prompt (`spu >`) in an *interactive mode.*

Diagnostic utilities are generally located in directory /mnt/bin.

Most of these utilities, or similar utilities with the same name, are used in both the C200/C3200 Series and the C3400 Series computers. Significant differences in the operation of the utility in both series are listed.

This manual documents the C3400 Series CPUs. Information about the C200/C3200 Series is included for comparison, only.

Man pages (online documentation) include information for all C Series systems to which a utility name applies.

NOTE

All diagnostic utilities within this chapter are offline in nature and should not be executed while ConvexOS is running.

Upon termination, each utility returns two bytes of status:

- One byte is supplied by the system, giving the cause for termination. This byte is 0 for normal termination.
- In the case of normal termination, another byte is supplied by the program. This byte is customarily 0 for successful execution, and nonzero to indicate troubles, such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously *exit code, exit status* or *return code*, and is described only where special conventions are involved.

bkplane(1d)

Backplane revision reporting utility

Syntax

```
bkplane
```

Function

Reads the backplane revision information from the scan ring and prints a message, including what type of backplane is installed and the assembly revision.

The messages for the backplane type are:

```
4 head Jav

8 head Jav

Jav Jr
```

The messages for the assembly revision are:

```
XX

A, B, ..., or Z

AA, AB, ..., or AR
```

Note

This utility applies only to C3400 Series machines.

clk_tune(1d)
Interactive clock tuning utility

Syntax

```
clk_tune
```

Function

Screen-oriented and interactive, this utility manipulates the clock tuning registers of the gate arrays on the JCPU processor board. Once the proper clock tune values are determined, you can save the tune values in the COP nonvolatile memory chip on the processor board. The tune values stored in the COP chip are used at system initialization time by the load_clk utility.

Notes

Running `clk_tune` corrupts the control stores of the JCPU processor board. The `cs` utility should be run after `clk_tune`.

This utility applies only to C3400 Series machines.

See also

load_clk
reset_cpus

commreg(1d)
Interactive communication register utility

Syntax

`commreg` [*anything*]

Function

Allows examination or modification of communication registers and their associated lock and modified bits (where applicable). All operations are performed by scanning and clocking the CPU utility board(s).

Normally, the `commreg` utility prints only the results of the requested operation. If it is invoked with any arguments on the command line, the verbose printing option is enabled, which allows the printing of additional information which may be useful in debugging a malfunctioning set of communication registers.

After invocation, the `commreg` utility prompts for a command. Each line is a separate command, and the input varies with the command, as indicated in the list of commands in either Table 39 or Table 40. All numbers must be entered in hexadecimal, *without* a leading `0x`. Communication register addresses may be specified in one of three ways:

- RAM address
- Physical address
- Logical address (on the basis of a CIR)

The logical address is entered with a communication index register (CIR) number (in hexadecimal), immediately followed by a colon (:), then the logical address for that CIR (in hexadecimal).

For the C3400 Series version:

Address type	Valid addresses (in hex)
RAM address	0—3FF
Physical address	3C00—3FFF
CIR	0—1F
Logical address	4000—401F, or 8000—803F

Table 39 lists the available commands for the C3400 Series version.

Table 39
`commreg` utility operations (C3400 Series)

Operation	Description
`w[rite]` *reg, udata, ldata, lock*	Write all 64 bits and lock bit
`r[ead]` *reg, udata, ldata, lock*	Read all 64 bits and lock bit
`d[ump | isplay]` [*reg* [*end_reg*]]	Display one or a range of registers and lock bits
`e[xit] | q[uit]`	Exit or quit

When multiple JCPUs are installed in a system, the operation affects the communication registers in each CPU. If the operation performs a read, each copy is read and compared. If there are no differences between the copies, just the data is displayed. Otherwise, a warning message is displayed, including the differences between copies.

For the C200/C3200 Series version:

Address type	Valid addresses (in hex)
RAM address	`0-3FF`
Physical address	`3C00-3FFF`
CIR	`0-7`
Logical address	`4000-401F`, or `8000-803F`

Table 40 lists the available commands for the C200/C3200 Series version.

Table 40
`commreg` utility operations (C200/3200 Series)

Operation	Description
`tst` *reg*	Display lock bit status.
`lck` *reg*	Set lock bit to 1.
`ulk` *reg*	Clear lock bit to 0.
`put_w` *reg l_data*	Write lower 32 bits.
`put_u` *reg u_data*	Write upper 32 bits.
`put_l` *reg u_data l_data*	Write all 64 bits.
`get_w` *reg*	Display lower 32 bits.
`get_u` *reg*	Display upper 32 bits.
`get_l` *reg*	Display all 64 bits.
`snd_w` *reg l_data*	Write lower 32 bits.
`snd_u` *reg u_data*	Write upper 32 bits.
`snd_l` *reg u_data l_data*	Write all 64 bits.
`rcv_w` *reg*	Display lower 32 bits.
`rcv_u` *reg*	Display upper 32 bits.
`rcv_l` *reg*	Display all 64 bits.
`wr_cmod` *reg mod_data*	Write associated modified bit.
`rd_cmod` *reg*	Display associated modified bit.
`rest` *reg u_data l_data lock_data*	Restore specified register.
`d[ump | isplay]` [*reg* [*end_reg*]]	Display one or a range of registers.
`e[xit] | q[uit]`	Exit or quit.

When performing communication register operations, DMODE is normally enabled. The `wr_cmod` operation, however, requires the communication registers to be clocked normally (without DMODE), so care should be exercised when writing modified bits. Otherwise, the state may not be preserved.

config_chk(1d)

Check and print system configuration

Syntax

```
config_chk [-c] [-f] [-m] [-t] [-u] [-v]
```

Function

Checks the boards installed in a system against wanted and required configuration constraints to determine if the configuration is valid.

Normally, the `config_chk` utility prints the current processor configuration (type, number, availability), then checks specific items, such as board combinations, to ensure compatibility. Warning or fatal error messages are printed if an illegal condition is present.

A negative number is returned if a fatal problem is encountered during execution.

Options

The following options are available:

`-c` Don't print the configuration information.

`-m` Don't check memory timing configuration.

`-t` Don't check service processor clock. Normally, the utility checks to ensure there is a reasonable time and date on the service processor clock.

`-v` Enter verbose mode; print what is being checked.

The following options apply only to the C200/C3200 Series computers:

`-f` Don't check for functional unit (FU) and data cache (DC) compatibility. Normally, the utility checks to ensure that all installed FUs and DCs are either enhanced functional units (EFUs) and enhanced data caches (EDCs), or scalar functional units (SFUs) and data cache units (DCUs). Class four machines are required to have EFUs and EDCs.

`-u` Don't check VPC compatibility. Either all 1205 VPCs or all 2205 VPCs are required.

cop(1d)
Display board identification information

Syntax

`cop [-v] [-e] [-m] slot [ slot]. . .`

Function

With no options, displays the slot name, board type, part number, serial number, and board revision of the boards in the specified slots. Proper board and slot matching is also checked, although the computers should not power up if any boards are installed incorrectly.

If a mismatch is detected, an error message indicating the mismatch is printed, and the `cop` utility returns an exit status of -1. Normally, `cop` returns an exit status of 0.

Options

The following options are available:

`-v` Board and slot matching is verified, but board identification information is not displayed.

`-e` Information about the manufacturing test level and missing assembly revisions is added to the standard board identification information.

`-m` Only the manufacturing test level code for the named slots is displayed. No other information is displayed. This option is intended for use in scripts.

slot Table 41, Table 42 and Table 43 contain mnemonics that may be used to specify slots.

Table 41 contains backplane-independent terms. Any mnemonic from this table may be used.

Table 42 and Table 43 contain slot names for each type of computer. Only mnemonics from the appropriate column of Table 42 or Table 43 may be used.

Table 41
Slot names (C Series)

Slot name	Description
all	All slots
mcm	All memory slots
mcme	All even memory slots
mcmo	All odd memory slots
mcm0	All memory 0 slots
mcm1	All memory 1 slots
mcm2	All memory 2 slots
mcm3	All memory 3 slots
cpu	All CPU slots
io	All I/O and support slots
pi	All PBUS interface slots
cu	All CPU utilities board slots
ccu	All CCU slots

Table 42
Slot names (C3400 Series)

Slot name	Description	Applicability
cpu0	CPU 0 slot	2, 4, and 8 CPU backplanes
cpu1	CPU 1 slot	8 CPU backplanes
cpu2	CPU 2 slot	2, 4, and 8 CPU backplanes
cpu3	CPU 3 slot	8 CPU backplanes
cpu4	CPU 4 slot	4 and 8 CPU backplanes
cpu5	CPU 5 slot	8 CPU backplanes
cpu6	CPU 6 slot	4 and 8 CPU backplanes
cpu7	CPU 7 slot	8 CPU backplanes

Table 42 (continued)
Slot names (C3400 Series)

Slot name	Description	Applicability
mo0	Memory 0 odd slot	2, 4, and 8 CPU backplanes
me0	Memory 0 even slot	2, 4, and 8 CPU backplanes
mo1	Memory 1 odd slot	4 and 8 CPU backplanes
me1	Memory 1 even slot	4 and 8 CPU backplanes
mo2	Memory 2 odd slot	4 and 8 CPU backplanes
me2	Memory 2 even slot	4 and 8 CPU backplanes
mo3	Memory 3 odd slot	4 and 8 CPU backplanes
me3	Memory 3 even slot	4 and 8 CPU backplanes
piy	PBUS Y interface slot	2, 4, and 8 CPU backplanes
pix	PBUS X interface slot	4 and 8 CPU backplanes
cuj	CPU utilities card slot	2, 4, and 8 CPU backplanes
ccu0	CCU 0 slot	2, 4, and 8 CPU backplanes
ccu1	CCU 1 slot	2, 4, and 8 CPU backplanes
ccu2	CCU 2 slot	2, 4, and 8 CPU backplanes
ccu4	CCU 4 slot	4 and 8 CPU backplanes
ccu5	CCU 5 slot	4 and 8 CPU backplanes
ccu6	CCU 6 slot	4 and 8 CPU backplanes

Table 43
Slot names
(C200/C3200 Series)

Slot names-2 CPU backplane	Slot names-4 CPU backplane	Description
mo0	mo0	Memory 0 odd slot
me0	me0	Memory 0 even slot
mo1	mo1	Memory 1 odd slot
me1	me1	Memory 1 even slot
mo2	mo2	Memory 2 odd slot
me2	me2	Memory 2 even slot
mo3	mo3	Memory 3 odd slot
me3	me3	Memory 3 even slot
pia	piy	PBUS Y interface slot
NA	pix	PBUS X interface slot
cpx	cpx	CPU utility card slot
ccu0	ccu0	CCU 0 slot
ccu1	ccu1	CCU 1 slot
ccu2	ccu2	CCU 2 slot
ccu3	ccu3	CCU 3 slot
NA	ccu4	CCU 4 slot
NA	ccu5	CCU 5 slot
NA	ccu6	CCU 6 slot
NA	ccu7	CCU 7 slot

Files

/mnt/usr/lib/DB_cop

cpureg(1d)

Initialize or display CPU nonvector register state

Syntax

`cpureg [-c # | all] [-i[nnnn]]`

Function

Initializes or displays (to stdout) the contents of the CPU nonvector register state.

Options

The following options are available:

`-c[#][,#]` Allows you to select which CPU(s) to initialize or display.
Default: All CPUs in the current system configuration.

When you select a specific CPU (#), the `cpureg` utility verifies it is in the current system configuration. Table 44 and Table 45 show the registers that are displayed.

`-i[nnnn]` Indicates the type of operation. If this option is not present, the `cpureg` utility *displays* the nonvector registers. If this option is present the `cpureg` utility *initializes* the following nonvector registers:

address registers	a0—a7
scalar registers	s0—s7
temporary registers	t0—t7

Selecting a constant value (*nnnn*) causes cpureg utility to initialize the registers to the specified value. Without an initialization value, a predefined pattern is written to the registers.

Following initialization, all the nonvector registers are displayed. C3400 Series registers are listed in Table 44. C200/C3200 Series registers are listed in Table 45.

Table 44
Registers displayed by `cpureg` (C3400 Series)

Register name	Description
pc	Program counter register
psw	Program status word register
upc	Micro program counter register
a0-a7	Address registers
s0-s7	Scalar registers
t0-t7	Temporary registers
vm_l	Vector mask register—lower half
vm_u	Vector mask register—upper half
vs	Vector stride register
global_int_enab	Global interrupt enable mask
local_int_enab	Local interrupt enable mask
ION	Interrupt on flag
RT_ION	Realtime interrupt on flag
CPUs present	Mask of CPUs in complex
RT_CPUs present	Mask of realtime CPUs in complex
broadcast enables	Broadcast enable register
RT broadcast enables	Realtime broadcast enable register

Table 45
Registers displayed by `cpureg` (C200/C3200 Series)

Register name	Description
pc	Program counter register
psw	Program status word register
upc	Micro program counter register
ccr	CPU control register
a0-a7	Address registers
s0-s7	Scalar registers
t0-t7	Temporary registers
vm_l	Vector mask register—lower half
vm_u	Vector mask register—upper half
vs	Vector stride register
global_int_enab	Global interrupt enable mask
local_int_enab	Local interrupt enable mask
ION	Interrupt on flag
int_mode	Interrupt mode
target_CPU	Target CPU to service global interrupts

Examples

Examples of the `cpureg` utility follow:

Example 1:

```
cpureg -c1
```

Displays elements of nonvector registers CPU 1.

Example 2:

```
cpureg -c0 -i10
```

Initializes nonvector register 0 in CPU 0 to the 64-bit value 10 (The absence of a leading `0x` on the initializing value causes `cpureg` to interpret the number as decimal).

Example 3:

```
cpureg -i0x123456789abcdef
```

Initializes all nonvector registers in each installed and available CPU to the hexadecimal value `123456789ABCDEF`.

Note

The `cpureg` utility only affects those CPUs you selected. No other subsystems should be affected by this utility.

See also

cpuvreg(1d)
reg_dump(3d)
regrd(3d)

cpuvreg(1d)

Initialize or display central processor vector register state

Syntax

`cpuvreg [-c#[,#]] [-v#[,#]] [-s#] [-n#] [-i[nnnn]]`

Function

Initializes or displays (to stdout) the contents of the C Series CPU vector register state.

Options

The following options are available:

`-c[#][,#]` Allows you to specify which CPU(s) (#) to initialize or display.
Default: All CPUs in the current system configuration.

When you select a specific CPU, `cpuvreg` verifies it is in the current system configuration. Table 46 shows the registers that are displayed.

`-v[#][,#]` Allows you to specify which vectors (#) of the vector processor to initialize or display.
Default: Process all vector registers for the specified CPU vector processors.

`-s[#]` Allows you to specify the starting element (#) within a vector register to initialize or display.
Default: Begin with element zero.

`-n[#]` Allows you to specify how many elements to initialize or display (#).
Default: Process all 128 elements of the vector.

`-i[`*nnnn*`]` Indicates the type of operation. If this option is not present, `cpuvreg` *displays* the vector registers. If this option is present `cpuvreg` *initializes* the vector registers.

Selecting a constant value (*nnnn*) causes `cpuvreg` to initialize the registers to the specified value. Without an initialization value, a predefined pattern is written to the registers. Following initialization, all the vector registers are displayed.

For C3400 Series, the display is of the form:

```
CPU [cpu#] [element#] Vx = IIIIII
```

Where

`IIIIII` Register content in hexadecimal.

For C200/C3200 Series information is displayed for the registers listed in Table 46.

Table 46
Registers displayed by `cpuvreg` (C200/C3200 Series)

Register name	Description
pc	Program counter register
psw	Program status word register
upc	Micro program counter register
ccr	CPU control register
a0-a7	Address registers
s0-s7	Scalar registers
t0-t7	Temporary registers
vm_l	Vector mask register—lower half
vm_u	Vector mask register—upper half
vs	Vector stride register

Examples

Examples of the `cpuvreg` utility:

Example 1:

```
cpuvreg -c1 -v0 -s3 -n2
```

Displays elements 3 and 4 of vector register 0 in CPU1.

Example 2:

```
cpuvreg -c0 -v0,3,4 -n2
```

Displays elements 0 and 1 of vector registers 0, 3, and 4 in CPU0 and CPU1.

Example 3:

```
cpuvreg -c0 -v0 -i10
```

Initializes vector register 0 in CPU0 to the 64-bit value 10 (The absence of a leading `0x` on the initializing value causes cpuvreg utility to interpret the number as decimal).

Example 4:

```
cpuvreg -i0x123456789abcdef
```

Initializes all vector registers in each installed and available CPU to the hexadecimal value `123456789ABCDEF`.

Note

The `cpuvreg` utility only affects CPUs selected.

See also

cpureg(1d)
vregrw(3d)

cs(1d)
Load the writable control store(s)

Syntax

`cs [-r] [-l] [-v] [-V] [-c#] [-e#] [-d[n1[n2]]] [-f] [-debug #] [-store_type ...]`

Function

Loads, verifies, and displays the contents of the C3400 Series writable control stores. The default for this command is to load and verify the control store images for all CPUs using the default file path names for each of the control store types.

The control store image files must be in the format generated by `hex2wcs`, the host utility which generates a control store loader data file from the hardware object file. The file name is the name of the store to be loaded, followed by the suffix .wcs. Typical file names are us.wcs or vd.wcs.

The `cs` utility first looks in the current directory for the store image files for each CPU. If they are not in the current directory, `cs` looks in the file /mnt/usr/ucode/cs_rev_info to determine the path to the store image files. Each line in this file is of the form

 # nonrt-directory rt-directory

where:

#	Part number for this board
nonrt-directory	Directory to obtain the control store image files for this board, if it is not a realtime CPU
rt-directory	Directory to obtain the control store image files for this board, if it is a realtime CPU

The `cs` utility uses the information in this file to load control images stored in the most efficient manner, reading the fewest number of files and doing as much as possible in parallel.

Alternate or test microcode may be loaded by creating a new directory and editing the appropriate line in the file /mnt/usr/ucode/cs_rev_info to point to that directory, or by invoking `cs` in a directory with microcode in it.

The high level order of execution is:

1. Parse the command line; abort if invalid options are encountered.
2. Read the necessary data files into local memory; abort if errors occur.

3. Initialize any necessary fields and information.
4. Loop writing data to all requested stores or CPUs, doing as much in parallel as possible.
5. Loop reading data from all requested stores or CPUs, doing as much in parallel as possible. If verification during reads is enabled, values are checked as they are read.
6. Loop verifying or displaying data from all requested stores or CPUs.

The options with the `cs` utility allow you to modify the program functions. If no options are specified, all stores are loaded on all available CPUs, using the default file names for each control store.

Options are processed left to right on the command lines, with those options further to the right taking precedence over those to the left for options that override functionality. Store specifications are cumulative.

Options

The following options are available:

`-c` Specifies the CPUs on which operations are performed. The default is all CPUs in the current system configuration. If any `-c` option is specified, stores are loaded only on those CPUs specified. All specified CPUs must be available. Only one CPU may be specified per `-c`, however multiple `-c`'s may be used, and their effect is cumulative.

`-d`[*n1*[*n2*]] Displays the contents of control stores. `n1` specifies the beginning address in decimal. `n2` specifies the ending address, if more than one is wanted. This option overrides both the load (`-l`) and verify (`-v`) options.

`-debug #` For internal use. The debug option must be followed by a positive number. The debug flag may be set to an internal debug level. The amount of information increases as the value gets higher. Suggested levels are one through five.

`-e` Specifies the number of errors to report during verification. The default for this option is to report five errors prior to aborting verification of the failing control store.

`-f` Specifies that the scan field names should be printed for any bits which fail to compare during verification. The default for this option is to not print field names. Specify this option to see field names.

`-l` Specifies load only mode. This mode does not verify the contents of the control stores after loading. This option overrides both the display (`-d`) and verify (`-v`) options.

`-r` Prints revision information only. This option may be used to determine the revisions of microcode loaded. CPUs are not affected.

`-v` Specifies verify only mode. This option compares the current contents of the control stores to the contents of the appropriate control store image files. This option overrides both the display (`-d`) and load (`-l`) options.

`-V` Specifies verify during reads. This option compares the expected and actual values during the read operation. A one line message indicating CPU, store, and address is displayed. The failing data is summarized in easy to read tables after all reads are done.

-store_type ... Allows you to specifically load, verify, and dump only the contents of the store(s) indicated by the option name(s) specified. The microcontrol store types are listed in Table 47.

Table 47
Writable control stores

Control store identifier	Description
us	Scalar microcode, three banks: uca, ucb, and ucc
ua	Vector add pipe microcode, two banks: 0 and 1
um	Vector multiply pipe microcode, two banks: 0 and 1
ul	Vector load/store pipe microcode, one bank
vd	Vector dispatch table, one bank
sr	Scratch RAM
ip	Instruction processor cracker RAMs

See also

hex2wcs(1d)
dump_wcs(1d)

dcache(1d)
Dump the data cache

Syntax

`dcache [-c # | all] [-m | -r] [-dh [n1[n2]]]`

Function

Halts the specified CPUs and dumps the contents of the data cache (to stdout) as either RAM addresses or a logical addresses, in hexadecimal format.

The default for this utility is to dump the entire data cache of all the CPUs in the current configuration in memory (logical) address mode.

If an error is detected, `dcache` returns an error code of -1. Otherwise, dcache returns an error code of 0. In RAM address mode, an error is reported if an address is larger than is supported in the hardware.

Options

The following options are available:

`-c[# | all]`	Selects a CPU (#) or all the CPUs in the current configuration. When you select a specific CPU, the dcache utility verifies it is in the current system configuration. Default: All CPUs.
`-m`	Specifies memory address mode (default). Addresses *n1* through *n2* are logical addresses.
`-r`	Specifies RAM address mode. Addresses *n1* through *n2* are physical or RAM addresses.
`-dh [n1 [n2]]`	Dump hexadecimal addresses *n1* through *n2* of the data cache. If no addresses are specified, the entire data cache is dumped.

Note

For C3400 Series:

- Only logical address bits <13..3> are significant.
- Only physical address bits <10..0> are significant.

For C200/C3200 Series:

- Only logical address bits <11..3> are significant.
- Only physical address bits <8..0> are significant.

See also

icache(1d) ipte_cache(1d)
pte_cache(1d) sram(1d)

diaginit(1d)
Diagnostics initialization script

Syntax

`.diaginit [-f]`

Function

The `.diaginit` shell script is invoked from `.profile` at boot time by SPU OS to initialize the scan ring description files. It also initializes the system configuration file /mnt/boot_db. This invocation is dependent upon three conditions:

- Power-on bit
- Changes in system configuration since last power cycle
- The user-supplied forced-initialization parameter (`-f`)

Options

The following option is available:

`-f` Allows forced initialization. Otherwise, initialization only occurs if both the power has been cycled off and the configuration of boards within the system has changed.

The `pup` utility examines the power-up bit in the soft front panel, and the `cop` utility determines the current set of board revisions.

If the conditions are set for scan ring initialization, then the `scnlink` utility builds a new composite scan definition in a SPU RAM shared memory buffer. The `mminit` utility is executed to determine the memory configuration so the `scn_util` utility can initialize the system configuration file /mnt/boot_db. The `pup` utility is executed to set the `power-on` bit to false (cleared). The file /mnt/usr/lib/initall_cpu is removed, if initialization is performed.

See also

pup(1d)
cop(1d)
mminit(1d)
ringrev_chk(1d)
scn_util(1d)
scnlink(1d)

Files

/mnt/usr/lib/initall_cpu
/mnt/errlog
/mnt/bin/config_chk
/mnt/usr/scn/cop.new
/mnt/usr/scn/cop.old
/mnt/boot_db

dshell(1d)
Diagnostic shell (test executive)

Syntax

dshell

Function

The CONVEX diagnostic shell (dshell) runs on SPU OS when ConvexOS is not running. *All CONVEX diagnostic tests, except standalone and online diagnostics, run under this shell.* This allows you to become familiar with one command interface and set of command options.

Syntax

The commands listed in Table 48 are available.

Table 48
dshell manual mode commands

Command	Description
`test`	Execute a test
`pause`	Enable or disable pause modes
`msgs`	Enable or disable message modes
`log`	Enable or disable logging modes
`loop`	Enable or disable loop modes
`status`	Print test flag status

`test` command

The `test` command causes a diagnostic test to be executed. The following options are available:

testname — Causes a specified test to be executed. If no *testname* is given, you are shown a menu of available tests and are asked to choose one. This menu is automatically generated the first time it is requested.

If no option is specified, then all subtests are performed once in ascending numerical order.

`-s` — Allows for selective execution of available subtests.

`-c` — Allows for selective execution of available classes.

The `-c` and `-s` options may not be mixed. Subtests or classes which are listed but do not exist are ignored.

If alone in the command line, the `-s` and the `-c` options cause a menu of all subtests or all classes in the chosen test to be displayed, with a prompt for your choice. If no option is specified, then all subtests are performed once in ascending numerical order.

The syntax for executing subtests or classes (via `-s` or `-c`) is the same whether entered in the command line, or as a response to a menu. When responding to a menu, the appropriate option (`-c` or `-s`) is assumed, and should not be included in the input stream.

The `-s` and the `-c` options allow for multiple and/or sequenced execution:

testname `-s n(i,j-k)` Execute *testname* subtests i and j-k n times

testname `-c n(i),j-k` Execute *testname* subclass i n times

Examples

Examples of selective subtests and classes follow:

Example 1:

```
test testname -s 10,15,5(20)
```

Executes subtests 10,15,20,20,20,20,20

Example 2:

```
test testname -s 3(10,15)
```

Executes subtests 10,15,10,15,10,15

Example 3:

```
test testname -c 2(13-10)
```

Executes classes 13,12,11,10,13,12,11,10

Note

Test input and output may be redirected via the command line as follows:

<*filename* Test input from *filename*

>*filename* Test output to *filename*

+>*filename* Test output to both *filename* and terminal

pause command

The `pause` command enables or disables pauses at specific points in a test. The following options are available:

`-f` [*nnn*]	Enables a pause on every failure, or at every failure in a specified subtest.
`-b` [*nnn*]	Enables a pause at the beginning of every subtest, or at the beginning of a specified subtest.
`-e` [*nnn*]	Enables a pause at the end of every subtest, or at the end of a specified subtest.
`off [-f] [-b] [-e]`	Disables all or specific pauses.
`continue` | **RETURN**	Continues the test from a pause. The continue command is a nop (no operation) command if not executed at a test pause.

When an executing test reaches a pause condition, it halts execution and displays the dshell prompt (`:`). You may then enter any dshell utility command or any SPU OS command via the `!` directive.

The default condition is

```
off -f -b -e
```

All pauses off.

msgs command

The `msgs` command enables or disables the pass/fail messages output by test programs. The following options are available:

`-f [long | short]`	Enables either long or short error messages.
`-s`	Enables subtest/class result messages.
`-t`	Enables test result messages.
`off [-f] [-s] [-t]`	Disables all or specific messages.

The default for these options is:

```
-f long -s -t
```

Long fail, subtest, class, and test messages are turned on.

log command

The `log` command enables or disables logging of multiple failures. The following options are available:

`-s` *nn*	Allows *nn* multiple failures per subtest.
`-t` *nn*	Allows *nn* multiple subtest failures per test.
off [-s] [-t]	Disallows all or specific multiple failures.

The default for these options is:

```
off -s -t
```

All multiple failures are turned off.

loop command

The `loop` command causes consecutive execution of a test or subtest. The following options are available:

`-s` *subtest*	Enables consecutive looping on the specified subtest.
`-t` *subtest*	Enables consecutive looping on the test.
`off [-s] [-t]`	Disables all or specific loop modes.
CTRL-C	Exits a loop and restores the command level to you, if no subtests are being executed. If any subtests are running, a menu of abort procedures is displayed. It does not reset either the `-s` or the `-t` switch.

The default for these options is:

```
off -s -t
```

All looping is turned off.

status command

The `status` command displays the current condition of all dshell test control options on the screen, and describes the effect each has in its current configuration.

exit commands

The following commands can be used to leave the dshell:

`exit`	Causes termination of all paused tests and exits the dshell.
`quit`	Waits for any currently executing or queued tests to complete execution, and exits the dshell.
CTRL-C	Restores the command level to you, if no subtests are being executed. If any subtests are running, a menu of abort procedures is displayed.
CTRL-B	Aborts the currently executing test and exits the dshell.

There is an important difference between `exit`, `quit`, **CTRL-C** and **CTRL-B**. Commands `exit`, `quit` and **CTRL-C** allow an executing test to finish executing *protected code*. **CTRL-B** kills the test regardless of the type of code currently executing. Thus, a **CTRL-B** can leave the machine in an undefined state, necessitating a reboot.

`help` command

The `help` command gives a brief syntax and command explanation of all dshell commands.

! directive

The ! directive is an escape to SPU OS, and allows for execution of one command which must immediately follow the !.

Example:

```
!mm
```

See also

libtest(3d)

For more information, refer to the "Diagnostic shell (dshell)" chapter in this book.

bugs

CTRL-C after a test command has been entered, but before the SPU has successfully forked the test process may not kill the test process.

errintd(1d)
Error interrupt daemon and logger

Syntax

`errintd` [`-e`] [`-h`] [`-s`] [`-t`] [`-c` *nn*] [`-r` *nn*] [`-f` *file*]

Function

Monitors a C Series computer system for hardware error conditions. Three classes of errors are detected by `errintd`:

- Environmental errors
- Soft errors
- Hard errors

Environmental errors are related to the system operating environment. Typical environmental errors include temperature out of range, loss of airflow, and excessive current consumption.

Soft errors are usually correctable errors (the error is transparent to the system user). Soft errors include single-bit memory system errors, a variety of peripheral interface adapter (PI2) transfer errors, C3400 Series cache tag parity errors, and C200/C3200 Series referenced and modified bits parity errors.

Hard errors include parity errors, internal references to nonexistent memory, and so forth. Hard errors always result in the immediate halt of the central processing unit (CPU) and are fatal.

In some cases, both environmental and soft errors can be fatal.

In addition to monitoring for errors, when started, `errintd` starts the main memory sniffer utility, `mm_sniff`.

In the normal operating environment, errintd utility is started by the operating system (SPU OS) utility `prtlog`. All output from `errintd` is time-stamped by `prtlog`, and copied to both the system console and the SPU file /mnt/errlog.

Single-bit memory error reports are not written to stdout. Instead, a record of these errors is maintained in the file /mnt/softlog. Single-bit memory errors are isolated to the memory chip level. A count of total soft errors for each failed memory chip is maintained. By default, `errintd` stores a maximum of 60 memory chip entries in the softlog file.

When the softlog file has reached 75% capacity and a burst of errors occur (at a rate of 1 every 10 seconds), the logging of new chips in error is throttled, or governed, to prevent the log from immediately reaching its capacity. Whenever throttling occurs, a message is written to the console.

When an environmental error is detected, the errintd utility continues to report the error once a minute until the error is corrected. An unattended environmental error can result in a hard error.

In the event of a hard error, `errintd` logs the error and exits. A copy of the last hard error is kept in the file /mnt/hardlog.

Options

The following options are available:

`-e`	Logs environmental errors
`-h`	Logs hard errors
`-s`	Logs soft errors
`-t`	Treats a JCPU soft error as a hard error (fatal)

If none of the first three options is specified (`-e`, `-h`, or `-s`), `errintd` defaults to logging all three types of errors. Soft error logging must be enabled for JCPU soft errors to be caught, regardless of whether they are to be treated as soft or hard errors.

`-c` nn	Specifies the maximum softlog size. The value *nn* is the maximum number of failed memory chips on which the softlog retains information. The default is 60 entries.
`-r` nn	Specifies the memory sniff rate in Mbytes/day. This option is passed to the `mm_sniff` utility. The default is 32 Mbytes/day. If a rate of zero is specified, or if soft errors are not being logged, the sniffer is not started.
`-f` *file*	Sends `errintd` output to *file*. Default: Output goes to stdout.

See also

hard_logger(1d)
mm_sniff(1d)
softlog(5d)

hard_logger(1d)
Hard error logger

Syntax

`hard_logger [-F] [-N] [-S] [-v] [-f` *file*`]`

Function

Isolates the source of a hard error. Soft errors can optionally be isolated by the `hard_logger`. Once a hard or soft error is detected, the `hard_logger` can be invoked to locate the specific type of error that occurred. However, the actual cause of the error is not necessarily isolated.

In any case, hard errors are always fatal and result in the immediate halt of the CPU. Hard errors include parity errors, internal references to nonexistent memory, channel control unit (CCU) bus errors, and others.

Upon invocation of the `hard_logger`, all system clocks are halted. This is necessary to scan the various scan rings throughout the system.

Options

The following options are available:

`-F` Scans all scan rings, regardless of the error source register (ESR) or soft error register (SEL) value. Normally the scan rings in a subsystem are not scanned if the ESR value or the SEL value doesn't indicate a pending error in that subsystem.

`-N` Signals the `hard_logger` it is being invoked noninteractively. This option changes how some errors are reported. When the hard logger is invoked interactively, it is not known if a hard error has actually occurred. When invoked noninteractively, it is always assumed that a hard error should be present. Only diagnostic tests should use this option.

`-S` Log soft errors also.

`-v` Specifies activate verbose mode. This option prints more information when errors are detected. In addition, it causes the `hard_logger` to print its version number when it is started.

`-f` *file* Sends output to *file*. Normally, the `hard_logger`'s output is to /mnt/hard_log, which is then displayed to stdout, if *file* is not specified. Three special file descriptors (*file*) are available:

#NN	Send output to file descriptor *NN*
-	Send output to stdout (default)
+	Send output to stderr

Bugs

Due to current SPU OS problems, the special file descripters provided with the `-f` option may not always work exactly as expected.

See also

errintd(1d)

hex2wcs(1d)
Convert data files for cs utility

Syntax

`hex2wcs` *infile outfile* `-v`

Function

Converts the ASCII (character) file (with .hex extension), formatted for the control stores from the hardware design group, into the binary files (with .wcs extension) needed by the C3400 Series control store loader.

The *infile* and *outfile* must be specified, in the order indicated.

Upon invocation of the `hard_logger`, all system clocks are halted. This is necessary to scan the various scan rings throughout the system.

Options

The *infile* and *outfile* must be specified, in the order indicated.

`-v` Specifies `hex2wcs` is to print the converted data. The default is not to print the data, which can be excessive for large stores.

Example

The following examples are currently used to generate the .wcs files for a C3400 Series system:

```
hex2wcs sram.hex sr.wcs
hex2wcs se_real0.hex us0.wcs
hex2wcs se_real1.hex us1.wcs
hex2wcs se_real2.hex us2.wcs
hex2wcs ic.hex ip.wcs
hex2wcs ux.hex ua.wcs
hex2wcs ux.hex um.wcs
hex2wcs ul.hex ul.wcs
hex2wcs vd.hex vd.wcs
```

See also

cs(1d)
dump_wcs(1d)

icache(1d)
Dump the instruction cache

Syntax

`icache [-c#] [-m|-r] [-dh [n1 [n2]] ]`

Function

Halts the specified CPUs and dumps the contents of the instruction cache (to stdout) as either RAM addresses or logical addresses, in hexadecimal format.

The default for this utility is to dump the entire instruction cache for all CPUs in the current configuration in memory (logical) address mode.

If an error is detected, the `icache` utility returns a -1. Otherwise, it returns a 0. In RAM address mode, an error is reported if an address is larger than is supported in the hardware.

Options

The following options are available:

`-c#`	Specifies a CPU (#) or all the CPUs in the current configuration. When you specify a CPU, the `icache` utility verifies that CPU is in the current configuration before attempting any operation. Default: All CPUs in the current configuration.
`-m`	Specifies memory address mode (default). Addresses *n1* through *n2* are logical addresses.
`-r`	Specifies RAM address mode. Addresses *n1* through *n2* are physical or RAM addresses.
`-dh [n1 [n2]]`	Dumps blocks *n1* through *n2* of the icache to stdout in hexadecimal format. If no block numbers are specified, the contents of the entire instruction cache are dumped.

Note

This utility applies only to the C3400 Series CPUs:

- Only logical address bits <12..3> are significant.
- Only physical address bits <9..0> are significant.

Files

/mnt/usr/ucode/opcode.hex
/mnt/bin/initall

See also

initall(1d) dcache(1d)
ipte_cache(1d) pte_cache(1d) sram(1d)

initall(1d)

Initialize the CPU control stores and main memory

Syntax

`initall [-c]`

Function

Initializes all control stores for each CPU installed in the current configuration, and all of main memory. The control stores are loaded from various microcode files in the directory /mnt/usr/ucode. See `cs` for details on control store loading.

Options

The following option is available:

`-c` Implies conditional initialization of the CPUs. If the file /mnt/usr/lib/initall_cpu exists, the CPUs are assumed to be initialized, and only memory initialization is performed. If /mnt/usr/lib/initall_cpu does not exist, both the CPUs and memory are initialized.

The `initall` utility with no arguments performs initialization on both the CPUs and memory.

If an error is detected during initialization, the `initall` utility aborts the initialization sequence. The `initall` utility returns a status of 0 for successful initialization and a status of 1 if the initialization fails.

Files

/mnt/usr/lib/initall_cpu

See also

cs(1d)
margin(1d)
mminit(1d)
sysreset(1d)

ipte_cache(1d)
Dump the internal PTE cache

Syntax

`ipte_cache [-c #] [-m | -r] [-dh [n1 [n2]]]`

Function

Halts the specified CPUs and dumps the contents of the internal PTE cache (to stdout) as either RAM addresses or a logical addresses, in hexadecimal format.

The default operation of the `ipte_cache` utility is to dump the entire internal PTE cache of all the CPUs in the current configuration in logical address mode.

If an error is detected, the `ipte_cache` utility returns a -1. Otherwise, it returns a 0. In RAM address mode, an error is reported if an address is larger than is supported in the hardware.

Options

The following options are available:

`-c #`	Specifies a CPU or all the CPUs in the current configuration. When you specify a CPU, the `ipte_cache` utility verifies that CPU is in the current configuration before attempting any dump operations. Default: All CPUs in the current configuration.
`-m`	Specifies memory address mode (default). Addresses *n1* through *n2* are logical addresses.
`-r`	Specifies RAM address mode. Addresses *n1* through *n2* are physical or RAM addresses.
`-dh [n1 [n2]]`	Dumps hexadecimal addresses *n1* through *n2*. If no addresses are specified, then the contents of the entire internal PTE cache are dumped.

Note

This utility applies only to C3400 Series CPUs:

- Only logical address bits <20..12> are significant.
- Only physical address bits <7..0> are significant.

See also

dcache(1d)
icache(1d)
pte_cache(1d)
sram(1d)

iscn(1d)
Interactive scan utility

Syntax

`iscn` [*script1, script2, script3 . . .*]

Function

Allows you to observe and control the internal states of individual central processor and memory boards. It does this through interactive display and alteration of bit values in scan rings located throughout the system. The `iscn` utility is intended for persons with a detailed knowledge of the internal workings of the system.

The `iscn` utility can operate in the program mode and in the screen mode. Each mode offers different capabilities to the user of `iscn` utility.

The `iscn` utility commands allow you to group, display, or manipulate fixed bit sequences to obtain bit status information for booting, diagnostics, fault isolation, and testing.

The `iscn` utility language supports input file redirection, as well as being able to include `iscn` files at any point during an `iscn` utility session. These commands can be grouped via scripts into functions that are invoked whenever the name of the script is entered (program mode). These capabilities allow the creation of a library of `iscn` utility scripts. Any number of separate scripts may be created and used.

The `iscn` utility also contains a screen-oriented interactive scan field editor that allows the display and modification of scan fields located throughout the system (screen mode).

The screen mode uses simple graphic representations of fields in horizontal columns, with numbers that indicate bit values in binary, hexadecimal, or even decimal. Interaction occurs with cursor control keys and control sequences to display and modify `iscn` utility field bits.

The program mode is used to predefine the format of a screen mode display. Screen specifications allow you to define the position of input and output (I/O) fields on the screen via a simple C-like screen description language. These I/O fields are used in conjunction with the edit command in the `iscn` utility.

The `iscn` utility is invoked by entering **`iscn`** at the `spu >` prompt, followed by any optional screen mode script names:

`iscn` [*script1,script2,script3 ...*]

This returns a header that contains initialization and update information on the iscn utility changes. The following prompt is then displayed:

```
iscn==>
```

Enter `help` or `?` for an on-screen list of `iscn` utility commands.

Table 49 is a comprehensive list of the `iscn` utility commands, with the short form or alias of each command, as well as the meaning of each command.

Table 49
`iscn` utility commands

Command	Alias	Meaning
!	NA	Executes an operating system command.
adjust	NA	Checks scannability of all specified rings.
bdrev	br	Checks or display the revision level of a board.
bdtype	bt	Checks the presence of a board type in a slot.
clear	clr	Clears the screen.
clock	c	Generates a clock pulse.
dump	du	Dumps scan ring save buffers to a named file.
edit	e	Invokes screen mode.
even_parity	NA	Generates even parity.
exit	NA	Leaves iscn utility.
fetch	NA	Fetches a field for placement in a register.
fprint	fpr	Prints to a file.
get	g	Gets a field value.
halt	hlt	Takes one or more boards out of the run state.
help	?	Displays help.
include	in	Reads a specified file as input to the interpreter.
iforce	NA	Forces a scn_rd operation on all gets.

Table 49 (continued)
`iscn` utility commands

Command	Alias	Meaning
`iupdate`	iu	Manipulates the iupdate flag or force output after puts.
`list`	NA	Lists the fields associated with the named scan ring.
`loadscan`	ls	Generates a scan command and a single clock pulse.
`log`	l	Creates a log file (this command cannot be logged).
`logl`	ll	Creates a log (this command can be logged).
`odd_parity`	NA	Generates odd parity.
`print`	pr	Displays a set of values on the screen.
`put`	p	Puts a field value.
`reset`	re	Resets a subsystem.
`restore`	rs	Restores a scan ring from a buffer.
`ritchie`	ri	Terse error messages issued when toggled on.
`run`	r	Puts a slot in the run state.
`save`	sv	Saves the state of a scan ring to a buffer.
`scnclear`	sclr	Clears a scan ring to all zeros.
`scnout`	so	Creates identical scan rings for multiple boards.
`screens`	sc	Prints out all names of defined screens.
`undump`	undu	Reads a named file to reconstruct a scan ring save buffer.
`verify`	v	Enables read or compare verification.

See also

For information about writing `iscn` utility scripts, refer to the CONVEX publication, *How to Write iscn Scripts*.

jcpu_custom(1d)
Reset the CPUs' clock tune values

Syntax

```
jcpu_custom
```

Function

Reads the text file /mnt/usr/lib/custom.txt to find a list of scan fields and values to be used in customizing the processor board initialization state. The text file data is processed by the `jcpu_custom` utility to produce the binary file /mnt/usr/lib/custom.dat.

The binary file is read when the CPU is initialized during a CPU test (`cpu4030, cpu4041,` and so forth), or by the `scn_util` utility when booting the operating system.

The `jcpu_custom(1d)` utility is run from the `.diaginit` script to ensure that the custom setup binary file /mnt/usr/lib/custom.dat is up-to-date. When the binary file is read by the initialization library routine, `jcpu_custom(3d)`, the file dates for the text and binary files are compared to ensure that the binary file is also up-to-date. If the text file is newer than the binary file, an error message is printed.

The text file /mnt/usr/lib/custom.txt has the following format:

```
#
# comment
#
scan_field    value    # comment
     .
     .
     .
```

Comments must be preceded by a `#`. A maximum of 1000 *scan_field* entries may be included. *value* may be specified in decimal, or preceded by `0x` for hexadecimal.

Any errors in the /mnt/usr/lib/custom.txt file cause the program to abort with an error message. Possible errors include illegal scan field names, invalid comments, or unrecognizable data values.

Note

This utility applies only to C3400 Series computers.

Files

/mnt/usr/lib/custom.txt
/mnt/usr/lib/custom.dat

See also

scn_util(1d)
sysreset(1d)

load_clk(1d)
Load the clock tune values

Syntax

`load_clk [-c #] [-v]`

Function

Reads the clock tune values from the COP chip of a C3400 CPU board and writes the values into the tune registers of the C3400 CPU gate arrays. Verifies the tune values for the SE, IP, and MI gate arrays and print any discrepancies.

The default operation is to read the tune values from the COP chip(s) for the CPU board(s), load the tune values into the gate arrays, and verify the SE, IP, and MI tune values for each CPU board that is installed.

If an error is detected, the `load_clk` utility returns a -1. Otherwise, it returns a 0.

Options

The following options are available:

`-c #`	Specifies the CPUs on which to perform the `load_clk` utility operations. When a specific CPU is requested, the `load_clk` utility verifies that CPU is in the current configuration before attempting any dump operations. Default: All CPUs in the current configuration.
`-v`	Allows you to verify the SE, IP, and MI gate array tune values without loading the values. The tune values are compared against the values read from the COP chip(s) on the C3400 Series CPU processor board(s).

Note

This utility applies only to C3400 Series computers.

See also

clk_tune(1d)
reset_cpus(1d)

man(1d)
Find manual information

Syntax

`man -k` *keyword*

`man -f` *filename*

`man [-] [-t] [`*section*`]` *title* . . .

Function

Gives information about the utilities used in diagnostic testing. Lists one-line descriptions of commands specified by name, or all commands whose descriptions contain the specified keyword. Provides online access to sections of the printed manual.

Options

The following options are available:

`-k` *keyword* — Prints a one-line synopsis of each manual section whose table of contents listing contains the specified keyword.

`-f` *filename* — Prints out the table of contents lines for related manual sections.

When neither `-k` or `-f` is specified, `man` outputs the specified manual pages. If a section specifier is given, the `man` utility looks in that section of the manual for the given titles. Section is an Arabic section number (for example 3). The number may be followed by a single-letter classifier. For example, `1d` indicates a diagnostic utility in section 1. If section is omitted, `man` searches all sections of the manual and prints the first section it finds, if any.

If the standard output is not a terminal, or if the - option is specified, `man` pipes its output through `cat` with the `-s` option to crush out adjacent blank lines. Otherwise it pipes its output through `more` to stop after each page on the screen. Press space bar to display the next screen of information.

Files

/mnt/man/cat

See also

more(1)
cat(1)

Bugs

The current version of `man` does not properly underline.

map(1d)
Display logical to physical mapping

Syntax

`map [-cnn] [-t` *nn*`]` *a1 a2 ... an*

Function

Displays the following information for a given logical address:

- The communication index register (CIR) for which the mapping is taking place
- The thread ID (TID) of the process
- The logical address being mapped
- The segment descriptor register (SDR)
- The contents of the first- and second-level page table entries (PTE1, PTE2)
- The thread-level page table entry (PTET) for PTE2s that indicate thread-level PTEs exist
- The physical address corresponding to logical addresses *A1 A2 ... An.*

Options

The following options are available:

`-c`*nn* — Allows you to select the CIR for which the logical-to-physical address translation should occur. Default: CIR 0.

`-t`*nn* — Allows you to specify the process thread to follow in addresses that use unshared data. Default: thread 0.

If the valid bit is not set for the SDR or a PTE, then question marks are printed for the remaining translation values.

Note

The `map` utility stops the clocks to the entire system in order to read the SDRs for the addresses specified. It then enables the clocks to the memory and I/O systems to perform main memory read operations via the EBUS.

margin(1d)

Read and set power supply and system clock margins

Syntax

C3400 Series:

`margin` [[`-q` | `-v` | `-f`] [`-n` | `-u` | `-l`] {*pwr-clk_name* [. . .]}
`margin` [[`-d`] [`-n` | `-u` | `-l`] [`clkccu` [# | `all`]]

C200/C3200 Series:

`margin` [[`-q` | `-v`] [`-n` | `-u` | `-l`] {*pwr-clk_name* [. . .]} [[`-x` | `-e`] [`clk`]]
`margin` [[`-d`] [`-n` | `-u`| `-l`] [`clkccu` [# | `all`]]

Function

Provides a means for the power supply and system clock margins to be read and set.

After setting the specified margin conditions, `margin` measures the resultant clock frequency and power supply voltages. Although not all power supplies can be margined, all the power supplies are monitored. If any clock or voltage exceeds its tolerance, `margin` displays a message indicating the clock or voltage in error, the current reading, and the acceptable tolerance limits. If no arguments are specified, the current margin conditions are displayed.

The `margin` utility returns a zero if all power supplies and clocks are within tolerance, and a nonzero if anything is out of tolerance.

Options

The following options are available:

`-q` Displays only those voltages or clocks out of tolerance.

`-v` Displays all voltages and clocks, regardless of tolerance conditions. If `-v` is the only argument, this display of conditions loops until **CTRL-C** is pressed. The mnemonics used in the display of voltage levels using the `-v` option are shown in Table 50.

Table 50
Voltage mnemonics used in `margin` display

Mnemonic	Description
`vr1`	Voltage reference 1
`vr2`	Voltage reference 2
`smb`	System monitor board

`-n|-u|-l` Sets the voltage and clock frequency options, as shown in Table 51.

Table 51
`margin` voltage and clock frequency options

Option	Effect on voltage	Clock rate
`-n`	Nominal voltage (typically +5 volts)	Nominal frequency
`-u`	Upper margin voltage (typically 105% of nominal)	Upper margin frequency (typically 110% of nominal)
`-l`	Lower margin voltage (typically 95% of nominal)	Lower margin frequency (typically 90% of nominal)

pwr-clk_name Specifies the power supplies and system clock (required). They may be margined individually, or in any combination, as shown in Table 52.

Table 52
`margin` power supply and clock mnemonics

Mnemonic	Description
`psp5`	+5 volt power supply
`psm2`	-2 volt power supplies
`psm45`	-4.5 volt power supplies
`psm52`	-5.2 volt power supplies
`ps`	All marginable power supplies present
`clk`	System clock
`all`	All of the above

`-d` Enables the `margin` utility to be used to margin VIOP clocks. The clocks for the microprocessor section of a VIOP are separate from the rest of the system. This margining requires the microprocessor section of the VIOP to be reset, and is a destructive operation. The mnemonic `clkccu` is followed a slot number (#) or the word all to specify which channel control units (CCUs) are to be affected by the command.

`-x` Sets the system clock (clk) to 50% of its nominal frequency (C200/C3200 computers only).

`-e` Sets the system clock (clk) to an external connector frequency (C200/C3200 computers only).

`-f` Forces all CPU clocks to the same frequency. Without this option specified, the `-x` and `-e` options will return all CPU clocks to their nominal frequencies. With this option specified before setting the clock rate, all CPU clocks will be set to the system clock rate.

Examples

```
margin -l clk -n psp5 -u psm2 psm45

margin -d -l clkccu all
```

mcm3_config(1d)

Determine memory board types and configuration settings

Syntax

`mcm3_config` [`-d`] [`-f` *bd_number conf_setting*] [`-p`] [`-v`]

Function

Checks the memory boards installed in a system and determines the scanned-in configuration setting for any MCM3 memory boards, in a typical invocation.

It may also be used to display the type of memory boards installed or force the configuration setting for an MCM3. Any MCM3's settings that are forced are not affected by a display setting operation in the same invocation. However, each specification operation increases the level by one, and these should be placed at the beginning of the command line.

`mcm3_config` returns a zero for a successful completion. It returns a nonzero if any problems were encountered.

Options

If no options are specified, the utility does nothing. The following options are available:

`-d`	Determines and sets the configuration for all MCM3s.
`-f`	Forces the setting of MCM3 parameters
bd_number	MCM3 board number (0-7)
conf_setting	MCM3 configuration setting (0-3)
`-p`	Prints the final configuration after all other operations are performed.
`-v`	Sets verbose mode. This is a debug option which can print a lot of information to the display, especially at higher levels.

memld(1d)

Load object file into system memory

Syntax

`memld file` [`-x`][`-o` *offset*][`-a` *physical_addr*]

Function

Loads an SOFF format object file into system memory. First-, second-, and third-level page table entries (PTEs) are set up for the file. The segment descriptor registers (SDRs) on the CPU utilities board(s) (CUJ) are initialized for CIR 0 to address the first-level PTEs.

Options

The following options are available:

`-x`	Specifies the page map file is *not cleared*. This allows the `memld` utility to be used repetitively to load files into memory without destroying the existing page tables.
`-o` *offset*	Allows a logical offset to be specified that is added to the logical addresses specified in the file (if the file is a relocatable format).
`-a` *physical_addr*	Specifies the starting physical memory address. The default load address is the first extant physical address in the system.

Hardware requirements

The memory system must be initialized before running this utility. In order to run, the system must contain at least a service processor, sufficient memory, a peripheral interface adapter (PI2), and a CPU utilities board (CUJ).

Hardware affected

The `memld` utility turns on clocks to the memory and I/O subsystems. These clocks are left on upon exit. No other clocks are modified. No subsystems are reset. The CUJ is scanned to load the values into the SDRs.

See also

mm_map(3d)
mem_load(3d)

mkdiag_db(1d)

Maintain diagnostics configuration file

enable_cpu

Enable installed CPUs

disable_cpu

Disable installed CPUs

Syntax

`mkdiag_db` [`-p`] [`-a`] [`-i`] [`-e` *keyword num type value*]

`enable_cpu` [`0, 1, 2, . . . 7`]

`disable_cpu` [`0, 1, 2, . . . 7`]

Function

Initializes the diagnostics configuration file, /mnt/diag_db. Examines the system hardware and software configuration to determine the appropriate default settings of the various parameters to be stored in the configuration file.

The utility commands `enable_cpu` and `disable_cpu` provide a means to modify the default configuration for installed CPUs. An installed CPU may be enabled or disabled by specifying the CPU.

Table 53 shows the system parameters determined by `mkdiag_db` for all C Series CPUs.

Table 53
System parameters determined by `mkdiag_db` (C Series)

Parameter	Description
PROCNUM	CPU population map (which CPUs are available)
OS_MODE	Set memory mode: 0=normal
INTRINSICS	Does machine-support microcoded intrinsic instructions
PARALLEL	Does machine-support parallel CPU processing
UNDER_MASK	Does machine-support the vector under mask instructions
SCALAR_ACCELERATOR	Specify scalar functional unit: 0=standard

Table 54 shows the system parameters determined by mkdiag_db for C200/C3200 Series computers:

Table 54
System parameters determined by mkdiag_db (C200/C3200 Series)

Parameter	Description
US_UCODE	Specifies microcode file name for us control store
UA_UCODE	Specifies microcode file name for ua control store
UL_UCODE	Specifies microcode file name for ul control store
UM_UCODE	Specifies microcode file name for um control store
VD_UCODE	Specifies microcode file name for vd control store
US_UCODE	Specifies microcode file name for us control store
SR_UCODE	Specifies file name for scratch RAM initialization

Options

The following options are available:

no options — A help message is printed.

-p — Prints the current configuration without making any changes to the configuration.

-a — Runs the automatic mode. Based on the hardware and software installed, the system configuration is determined. It is assumed that you want to take advantage of any available upgrades.

-i — Runs the interactive mode. It prompts you for all available options in configuring the system. This option is useful to set up a machine in a nonstandard configuration.

`-e` *keyword num type value*

Edits the database entry specified by *keyword*.

The data element number specified by *num* is replaced with the value specified by the argument *value*. The *value* argument may be numeric or alphanumeric.

The argument *type* specifies whether the value is a decimal number (`d`), hexadecimal number (`x`), or an ASCII string (`s`).

Notes

When any change is made to the diagnostics configuration file, /mnt/diag_db, by running `mkdiag_db`, or by entering **`enable_cpu`**, or **`disable_cpu`**, the changes must be reflected in the system boot configuration file, /mnt/boot_db. This is done automatically by running the `.diaginit` script or the `boot` utility. It can also be done manually by entering

```
scn_util -b /mnt/boot_db
```

Files

/mnt/diag_db

See also

diaginit(1d)
scn_util(1d)

mm(1d)
Display or modify main memory

Syntax

mm [-s] [sdr0] . . . [sdr7]

Function

Displays and modifies main memory on C Series computers and the population configuration map (PCM) on the service processor.

If segment descriptor registers (SDRs) are specified on the command line, these SDRs are used for logical-to-physical address translations. If no SDRs are specified, all SDRs are read from the utility board. This involves stopping clocks in the entire system while the SDRs are read. If any SDR is specified on the command line, mm does not read any SDRs from the utility board.

mm initializes no subsystems.

Options

The following options are available:

-s	Keeps the service processor from altering the clocks in the system. If this option is not present, then the mm utility turns on all clocks in the memory and I/O subsystems, and turns off the clocks in all other subsystems.
[sdr0] . . . [sdr7]	If segment descriptor registers (SDRs) are specified on the command line, these SDRs are used for logical-to- physical address translations. If no SDRs are specified, then all SDRs are read from the utilities board(s). This involves stopping clocks in the entire system while the SDRs are read. Specification of *any* SDR on the command line keeps the mm utility from reading SDRs from the utilities board(s). Note that the mm utility initializes no subsystems.

When the mm (*CIR#,TID#*): prompt is displayed, this utility is ready for command input.

CIR#	The communication index register that locates SDRs used during logical-to-physical address translation.
TID#	The current thread ID number used for logical-to-physical address translation.

Commands are of the general form:

```
command parameters
```

Unless noted otherwise, all numeric values are in hex. All address values represent either logical or physical addresses as specified by the `s` command. Use **CTRL-C** to abort an operation and return to the mm utility prompt. The following commands are available:

`b` Displays the current memory usage. An example output of the command is illustrated in Figure 65.

Figure 65
Example output of **mm b** command

```
File#  Physical Address   Pid   File Name     Logical Offset
  1    00000000-00014fff  120   p0r0_4331      00000000
  2    00015000-0006afff  120   cpu4331.rnn    00020000
  3    0006b000-0006dfff  120   support_4331   9ffff000
  4    0006e000-00077fff  120   p0rN_4331      20000000
  5    00078000-00081fff  120   p0rN_4331      40000000
  6    00082000-0008bfff  120   p0rN_4331      60000000
 --    00ff9000-00ffefff  120   pte2            NA
 --    00fff400-00ffffff  120   pte1            NA
```

The display lists a file number used by the `ln` command.

The physical address range shows the locations occupied by the specified file in main memory.

The `Pid` field is an arbitrary number assigned by the test program during loading, all files with the same Pids share a common set of logical address to physical address mappings, that is, they have the same SDRs.

The logical offset is the relocation offset added to the addresses in the file at the time it was loaded.

In order to use the symbolic features of the `mm` utility, it is necessary to load the symbol table at the correct offset (refer to the `l` and `ln` commands). The reserved file names pte1, pte2, and ptet are the locations where the page tables are in main memory.

This information is kept in the file /mnt/test/CPU/page_map that is used by most CPU tests. If this file does not exist, the b command displays the message:

```
/mnt/test/CPU/page_map does not exist.
```

`d` *a1* [*a2*] — Displays the contents of locations *a1* through *a2* in hex and ASCII.

`d` *a1,count* — Displays the contents of locations *a1* through *a1+count* in hex and ASCII.

`dp` — Displays the contents of the service processor PCM.

`e [on | off]` — Enables or disables error detection and correction (EDC) for the duration of the mm utility.

`f` *a1 a2 patt* — The hexadecimal pattern *patt* is repetitively copied into locations *a1* through *a2*.

`h` — Displays a command summary.

`i` *a1 a2* — The memory starting at location *a1* through location *a2* is disassembled and displayed in mnemonic form.

`i` *a1,count* — The memory starting at location *a1* for *count* instructions is disassembled and displayed in mnemonic form.

`m` [*a1*] — The memory modification mode is entered at address *a1*. If no address is specified, the default address is zero. Memory modification is performed on a byte basis. The display has the following format:

`addr:` *old_data* [*new_data*]opc

The data at a particular location is changed by entering new data and terminating the data value with an operation code. If *new_data* is omitted, the *old_data* is unchanged. The operation codes consist of the following:

RETURN	Increment address
-	Decrement address
=	Display same address
g	Go to address specified by *new_data*; do not modify *old_data*
q	Return to the mm utility prompt

`p` [*a1*] The PCM modification mode is entered at block *a1*, where *a1* is hex. If no block address is specified, the default address is zero. The PCM modification mode uses the same operation codes as the memory modification mode.

`q` Exits the `mm` utility.

`l` *filename* [-O*offset*] [*id#*]
Loads the symbols from the file name and adds the specified offset to each of the symbols. An identification number is optional, and is used only when the same file is going to be loaded multiple times. This identification number is a means to distinguish between two different symbol tables created from the same file.

`ln` *filenumber* Loads the symbols from the file specified by *filenumber*. *filenumber* is the number of the file listed when the `b` command is used. The symbol table offset is obtained from the page_map file. The default id# associated with the symbol table is 1.

`s [log] [phys]`
Sets the addressing mode of the `mm` utility. Logical mode assumes that logical addresses are used, and performs logical-to-physical address translations on all addresses. The SDRs used are either those specified on the command line, or those that exist for the specified CIR.

`ps` Pushes (rotates) the symbol tables located on the symbol table stack. When a symbol is used, it is located by searching the tables in the order specified on the symbol table stack, where the first occurrence of a symbol is used. This command can be used to ensure that the correct instance of a symbol is located.

`c` *num* Changes the CIR value. When the CIR value is changed, the new SDRs are always obtained from hardware. The new SDRs are then used for logical-to-physical translation.

`t` *num* Sets the thread ID (TID), which is then used during all subsequent logical-to-physical address translations. The thread ID is only significant in translating addresses located in memory that has been mapped as being threaded.

`!` A shell is invoked to interpret and execute the remainder of the line following the `!`. Following the shell command, system clocks are restored to the state that `mm` sets them to at startup unless `mm` was invoked with the `-s` option.

The following three I/O commands are analogous to similar commands above, and apply only to the C200/C3200 Series computers:

`d` *a1* [*a2*] Displays the contents of I/O locations *a1* through *a2* in hex and ASCII.

`d` *a1,count* Displays the contents of I/O locations *a1* through *a1+count* in hex and ASCII.

Note that only the most significant 2 bytes of a longword are meaningful in I/O space, so *xx* is printed for the low 6 bytes of each longword to avoid confusion.

`m` [*a1*] The I/O modification mode is entered at address *a1*. If no address is specified, the default address is zero.

I/O space modification is performed on a halfword (16-bit) basis, and only the most significant halfword of each longword is meaningful. Therefore, only longword-aligned addresses can be modified. The operating codes in this mode are the same as those available to the `m` command described previously.

Hardware requirements

Initialize the memory system before running this utility. The system must contain at least a service processor, two memory boards, and a peripheral interface adapter to run. If SDRs are to be read from the system, then a utilities board must also be installed.

Hardware affected

Upon exit, the `mm` utility leaves the memory and I/O system clocks running, if the `-s` option is not specified. If the `-s` option is specified, all clocks are as they were upon entry. The `mm` utility performs no scan operations, unless it is necessary to read the SDRs from a utilities board or the enable or disable EDC command is executed.

Bugs

After power up, the memory system must be reset with the `sysreset` utility before the `mm` utility can access main memory. Additionally, the PCM and main memory must be initialized by the `mminit` utility before other utilities can access main memory.

mminit(1d)
Main memory initialization

Syntax

`mminit [-c n] [-m] [-p n] [-s] [-i 8|16|32] [-f] [-P]`

Function

Initializes main memory after system power up. Normally, sets up the interleave and mode information, sizes main memory, initializes all the population configuration maps (PCMs), generates the file /mnt/usr/lib/DB_mcm, clears all the locations in main memory, and displays a table of the memory blocks found to be present. The default initialization mode utilizes the vector processing capability of the central processor. If an error is detected during initialization, `mminit` returns a -1. Otherwise, it returns a 0.

If both /mnt/usr/lib/DB_mcm and /mnt/boot_db exist, and `mminit` is able to correctly read the PCM entry from /mnt/boot_db, then `mminit` uses the PCM obtained from the file. If either of the files do not exist, `mminit` sizes memory. Memory is sized by initializing portions of memory from the top of memory to the beginning of memory then reading and checking the initialized areas to determine the amount of memory in each bank of each memory slot. From this information `mminit` generates the file /mnt/usr/lib/DB_mcm. `mminit` then initializes all system PCMs, including PCMs located on the service processor, utility and peripheral interface adapter boards.

`mminit` also initializes the configuration registers for the current interleave factor in use. This includes interleave registers located on the service processor, peripheral interface adapter, and the data cache unit (DCU) of each CPU. `mminit` determines the maximum allowable interleave by comparing the relevant PCM entries for the slots. The general rule of thumb is for all the board pairs to have the same memory sizes in order to maximize the interleave factor. The default interleave factor is 8-way interleave. The following checks are made:

- If slot pairs one and two or slot pairs three and four are present and have identical PCM entries, then 16-way interleave is allowed.
- If slot pairs one through four are present, and all four have identical PCM entries, then 32-way interleave is allowed. However, if slot pairs one through four are present and either slot pairs one and two or three and four do not have identical PCM entries, then the interleave is forced back to 8-way interleave.
- If slot pairs one through three are present and slots one and two have identical PCM entries, then 16- plus 8-way interleave is allowed.

Options

The following options are supported:

`-c` *n* — Initialize main memory using CPU *n*. The default is the first CPU found in the current configuration.

`-m` — Initialize main memory only. It is assumed that main memory has already been sized and the PCM initialized. Refer to the note on the `-f` option.

`-p` *n* — Set the memory initialization pattern to the longword hex pattern specified by *n*. The default longword pattern is zero.

`-s` — Use the service processor to perform the initialization via the EBUS. No vector operations are performed, and the CPU is not involved in the initialization. This mode of initialization is significantly slower than the default vectorized initialization mode.

`-i [8|16|32]` — Force the interleave factor. This mode allows you to bypass the normal method of initializing the interleave register to the maximum interleave. No error checking is performed to verify that a valid interleave is being selected.

`-f` — Force the PCM initialization via memory poking. This option overrides the ability to read the PCM from the /mnt/boot_db file. It should be noted that this option should not be used in conjunction with the `-m` option. The `-m` option takes precedence over this option.

`-P` — Perform PCM initialization only. The `mminit` utility determines the PCM (either from the /mnt/usr/lib/DB_mcm file or via memory poking) for all system PCMs and returns.

Diagnostic messages

The mminit utility has the following diagnostic messages:

```
scn_init failed
```

The mminit utility was unable to initialize the scan machine in order to perform its function.

```
Error trying to remove /mnt/test/CPU/page_map
```

The mminit utility received an error when it tried to remove the page map file.

```
mminit: undefined machine class XX
Using class YY as default
```

The machine class from the backplane was an unknown type. The mminit utility defaults to a C130 configuration in this case.

```
No CPU available. Forcing -s option
```

The mminit utility could not find a CPU to execute the initialization and is forcing an initialization from the SPU.

```
mminit: No memory boards found
```

The mminit utility didn't find any memory board pairs.

```
mminit: slot XX and slot YY are not populated the same (mcm)
```

The mminit utility did not find an MCM in both the odd and even slots.

```
mminit: continuing using internal PCM
```

The mminit utility had an error initializing a PCM and is using what it considers the correct PCM.

```
Slot XX and slot YY have different memory sizes.
Unable to interleave across them
```

The mminit utility was not able to upgrade the interleave due to memory incompatibilities.

```
Can not upgrade to 16 way interleave due to mismatch on
pairs XX and YY
```

The mminit utility was unable to go to 16-way interleave because one of the four slot pairs is incompatible.

Error messages

These error messages relate to the loading and execution of CPU code:

`Error during memory load`

The `mminit` utility was unable to load the CPU code into main memory for execution.

`Error during read of memory address map`

The `mminit` utility received an error when it tried to read the addresses of the relevant data areas in main memory.

`Invalid mminit.x00 request code`

The CPU expects one of the two types of request codes. For this case, the CPU received a type from the service processor that it did not expect.

`Hard error occurred`

The service processor code detected a hard error during the CPU codes execution.

`CPU time out`

The CPU did not complete execution in the time allowed by the service processor code.

`Vector valid trap`

A vector valid trap was detected by the CPU code.

`Unknown CPU error type: XXXX returned in PDT`

The CPU returned an invalid error type to the service processor when it completed.

`Error exit`

The CPU took an error exit due to the execution of an all zero opcode.

`Undefined opcode`

The CPU detected an undefined opcode during execution.

`Invalid process exception code XXXX`

The CPU took a process exception.

Subsystems affected

The `mminit` utility affects the execution of code running on any of the subsystems since it initializes main memory that is used by all the subsystems.

Bugs

The force interleave option does not allow the 16+8 interleave option to be used. This can be accomplished by using the `sp2util` utility to modify the PCR register of the SPU and then performing a `sysreset` to force the interleave register to migrate to the PIA and DCU.

mm_sniff(1d)
Main memory sniffer

Syntax

`mm_sniff` [`-r` *nn* | `-t` *nn*]

Function

Reads all main memory within a specified amount of time. The intention is to detect locations with a single-bit error. Once detected, the `errintd` utility can attempt to eliminate the error by performing a memory scrub operation on the location in error.

If a location contains a single-bit error and is not read for a long period of time, it could potentially drop another bit. This would result in a double-bit error that is not correctable. In addition, multiple-bit errors are hard errors, which halt the CPU.

The memory sniffer always reads main memory in four-page groups (for example, 16 kbytes, 1/64 the size of one population configuration map (PCM) entry). Upon invocation, based on the sniff rate in Mbytes per day, the `mm_sniff` utility calculates the number of seconds to sleep between each read. In the event the calculated sleep time is less than 15 seconds, the value is forced to 15 seconds. This time and the time required to sniff the entire memory system are reported.

Options

The following options are available:

`-r` *nn*	Set sniff rate to *nn* Mbytes/day. Default: 32 Mbytes/day.
`-t` *nn*	Set sniff sleep time to *nn* seconds. This option overrides the `-r` option.

See also

errintd(1d)
softlog(5d)

pte_cache(1d)
Dump the PTE cache

Syntax

`pte_cache [-c # | all] [-m | -r] [-dh [n1 [n2]]]`

Function

Halts the specified CPUs and dumps the contents of the internal PTE cache (to stdout) as either RAM addresses or logical addresses, in hexadecimal format.

The default operation of the `pte_cache` utility is to dump the contents of the entire PTE cache of all the CPUs in the current configuration in logical address mode.

If an error is detected, the `pte_cache` utility returns a -1. Otherwise, it returns a 0. In RAM address mode, an error is reported if an address is larger than is supported in the hardware.

Options

The following options are available:

`-c # | all`	Specifies a CPU (#) or all the CPUs in the current configuration. When a CPU is requested, the pte_cache utility verifies that CPU is in the current configuration before attempting any dump operations. Default: All CPUs in the current configuration.
`-m`	Specifies memory address mode (default). Addresses *n1* through *n2* are logical addresses.
`-r`	Specifies RAM address mode. Addresses *n1* through *n2* physical or RAM addresses.
`-dh` [*n1* [*n2*]]	Dumps addresses *n1* through *n2* of the PTE cache. If no addresses are specified, then the contents of the entire PTE cache are dumped.

Note

For C200/C3200 Series and C3400 Series:

- Only logical address bits <21..12> are significant.
- Only physical address bits <9..0> are significant.

See also

dcache(1d)
icache(1d)
ipte_cache(1d)
sram(1d)

pup(1d)
Power-up bit read and write utility

Syntax

`pup [? | on | off ]`

Function

Reads and writes the power-up bit on the service processor.

Options

The following options are available:

no options	The current state of the power-up bit is returned.
`on | off`	The power-up bit is set to 1 or 0, and the previous value is returned. If an error occurs, -1 is returned.
`?`	Displays the command format for invoking the `pup` utility and returns a -1.

See also

ioaccess(2d)
signal(2)
cop_read(3d)
cop_write(3d)

Bugs

Due to the sensitivity of the cop functions, reading and writing the cop chip are indivisible operations to you. Aborting the `pup` utility during either of these operations is impossible. If the operation of the cop hangs, then the `pup` utility operation (and the terminal) is hung.

reset_cpus(1d)

Reset the CPUs' clock tune values

Syntax

`reset_cpus [debug]`

Function

Resets the C3400 Series CPU clock tune values to a known state so that the board can be scanned. Normally, runs only at powerup from the `.diaginit` script.

Options

The following option is available:

`debug`	Enables debug output

Note

This utility applies only to C3400 Series computers.

See also

load_clk(1d)
clk_tune(1d)

scnlink(1d)

Intermediate scan ring definition file linker

Syntax

`scnlink` [`-c` *cop_file*] [`-o` *output_file*] [`-p` *old_file*] [`-d` *directory*] [`-f` *number*]

Function

In normal use on a service processor, examines the file /mnt/usr/scn/cop.out to determine the revisions of boards installed in a system, and sets up a number of data structures in shared memory. Without the initialization of these data structures, any utility or test that uses scan cannot be executed. Thus the `scnlink` utility must be executed on a service processor before most utilities or tests will work.

The `scnlink` utility returns a 0 upon successful completion, 1 otherwise.

Options

The following options are available:

`-c` *cop_file*	Use *file* instead of /mnt/usr/scn/cop.out.
`-o` *output_file*	Produce a specified *output_file* in the old `scnlink` format. This option is intended for debugging or host use, as special code is required for tests and utilities to be able to make use of this file. Shared memory is not modified when this option is specified.
`-p` *old_file*	Usable only in conjunction with the `-o` option. If the *output_file* specified with the `-o` option exists it is moved to *old_file* before the new file is created. Any existing *old_files* are removed.
`-d` *directory*	Use *directory* instead of /mnt/usr/scn as the directory to find the scan ring revision files.
`-f` *number*	Force the system serial number to be *number*, which must be entered in hexadecimal, without a leading `0x`.

The approximate order of execution is:

1. Ensures correct version of SPU OS
2. Parses arguments and verify combinations
3. Sets up the scan ring structures
4. Does any machine specific modification of these structures
5. Parses the cop.out file to see what is installed
6. Reads the scan ring revision files for the installed boards
7. Gets information about shared memory buffer (if needed)
8. Relocates and combines ring revision files
9. Writes the resident buffer (if it is to be written and enough space is available)
10. Writes the output file (if it is to be written)

Note

Even though the `scnlink` utility has changed significantly due to the changes made in 5.1 SPU OS, backward and host compatibility remain.

See also

scn(3d)
cop(1d)

Files

/mnt/usr/scn/cop.out
/mnt/usr/scn/[ringname]_rev[number]

scn_ring(1d)

Interactive read, write, and check scan ring utility

Syntax

```
scn_ring
```

Function

Allows you to interactively read, write, or check a scan ring. It provides you with available and default options at each prompt. It also checks all input for errors.

You are first requested to enter a scan ring name. Then the `scn_ring` utility repeatedly prompts you for a command. You can specify any of the commands.

Commands

The following commands are available (after ring specification):

`print ring information`
: Prints information about scan rings in general, and the specified ring in particular.

`read ring`
: Prompts for the direction and number of bits to read, and then prints the data read (if read printing enabled).

`write ring`
: Prompts for the direction and number of bits to read, background value, and foreground bit to set (if any), and then prints the data written (if write printing enabled).

`spu4000 class 2 subtest`
: Tests the ring with the applicable `spu4000` class 2 subtest.

`spu4000 class 4 subtest`
: Tests the ring with the applicable `spu4000` class 4 subtest.

`compare last read data with last write data`
: If the last read and write operations were of the same ring, the data is compared and any miscompares are displayed.

`enable write ring image printing`
: Toggles on or off the display of the scan ring buffer on ring writes.

`enable read ring image printing`
: Toggles on or off the display of the scan ring buffer on ring reads.

`select a different ring`
: Goes back to the ring specification prompt.

`exit`
: Exits (quits) the `scn_ring` utility.

scn_util(1d)
Hardware initialization utility

Syntax

`scn_util [-t] [-i] [-I] [-b] [-j` *addr_mode*`] [-s sdr0 ... sdr7]`

Function

Used by the CONVEX operating system to initialize hardware and check status. Several options are available, most of which should be used one at a time, since some are mutually exclusive and others make little sense if used in combination.

Options

The following options are available:

`-t` Treats JCPU soft errors as hard errors. Certain JCPU errors are switchable between being hard or soft errors, and are normally initialized by `scn_util` to be soft errors. This flag overrides this and makes them considered hard errors.

`-i` Initializes the clocks for memory and the I/O system. After this option is used, memory can be read or written from the SPU, and CCUs have their clocks enabled. The `-i` option returns an exit code of 0.

`-I` Returns the state of the clocks in memory and the I/O system. If these clocks are enabled, a 0 status code is returned. If the clocks are not enabled, a nonzero status code is returned.

`-b` This option outputs the system configuration database to stdout.

The format of stdout has information regarding the current system configuration, and consists of multiple `IDENTIFIER`s that are terminated by a semicolon. Each `IDENTIFIER` entry is as shown here:

```
IDENTIFIER(entry count,entry size,format) entry0,entry1,...entryN-1;
```

`IDENTIFIER` is an ASCII mnemonic describing some aspect of the system.

The `entry count` describes the number of entries that follow the closing parenthesis.

The `entry size` is the number of bytes required to hold each entry.

The `format` is the format of the entry: either `d` for decimal data, `x` for hexadecimal data, or `s` for string data (terminated by ;).

Table 55 lists the currently supported IDENTIFIERs for C Series.

Table 55
System identifiers
(C Series)

Identifier	Description
CPUTYPE	CPU instruction architecture: 1=C1, 2=C210a, 3=C200 Series, 4=C3400 Series
MEMSTART	Starting physical address of memory
IOPS	Location of IOPs in CCU slots
VIOPS	Location of VIOPs in CCU slots
IOP_ACCEL	Location of IOPs with accelerate/enable option
HSPS	Location of HSPs in CCU slots
IEEE	Does machine support IEEE mode arithmetic
PCM	Physical configuration map (each bit = 2 Mbyte memory block)
SERIALNUM	Serial number
MACHCLASS	Machine class: 0=C1, 1=C1XE, 2=C210,C220, 4=C230,C240, 5=C201/C202, 6=C232, 7=C3400, 8=C3400J
PROCNUM	CPU population map (which CPUs are available)
OS_MODE	Set memory mode: 0=normal, 1=V6.2 mode
INTRINSICS	Does machine support microcoded intrinsic instructions
PARALLEL	Does machine support parallel CPU processing
UNDER_MASK	Does machine support the vector-under-mask instructions

Table 56 lists currently supported `IDENTIFIERS` for C200/C3200 Series only.

Table 56
System identifiers
(C200/C3200 Series)

Identifier	Description
SCALAR_ ACCELERATOR	Specify scalar functional unit: 0=standard
US_UCODE	Specifies microcode file name for `us` control store
SR_UCODE	Specifies file name for scratch RAM initialization
UA_UCODE	Specifies microcode file name for `ua` control store
UL_UCODE	Specifies microcode file name for `ul` control store
UM_UCODE	Specifies microcode file name for `um` control store
VD_UCODE	Specifies microcode file name for `vd` control store

`-j` *addr_mode*

Enables the CPU to start execution at the indicated address. If the mode is 0, execution begins immediately. If the mode is nonzero, a set of prompts appears so that certain modes can be modified prior to starting the CPU.

The only mode selection possible at this time is the state of the data cache (dcache).

An exit code of 0 is returned if the start is successful. A nonzero value is returned if the start is unsuccessful.

`-s sdr0 ... sdr7`

Allows setting the segment descriptor registers (SDRs) to the specified values. There must be eight SDRs specified, otherwise an error is returned.

An exit code of 0 is returned if the operation is successful. A nonzero value is returned if the operation is unsuccessful.

Files

/mnt/bin/scn_util

secure(1d)
Enable or disable the secure mode

Syntax

`secure [on | off]`

Function

Enables or disables the secure mode of the microcode.

Options

The following options are available:

`on`	Enables the secure mode
`off`	Disables the secure mode

This command must be executed before the CPU is initialized. Once the CPU is running, this command has no affect.

The `.diaginit` script should be run after changing the state of the secure mode. See diaginit(1d).

Operation

This command modifies an entry in the /mnt/diag_db database. Subsequent initialization routines look at the value in this database to determine how to perform initialization. The enabling of the secure mode of the microcode causes additional state to be cleared when a head transitions from idle. This causes a slight performance degradation.

Note

Running in secure mode should have no other effect on user programs. The secure mode slightly alters the implementation of the C Series architecture, such that implementation verification routines such as some of the functional CPU tests operate differently when running with the secure microcode.

Files

/mnt/diag_db

See also

diaginit(1d)
mkdiag_db(1d)

sfpread(1d)

Read and modify the SPU front panel switches

Syntax

`sfpread [-i] [-v] [field_name]`

Function

Reads and modifies the contents of the SPU front panel. It operates in two different modes.

`-i`	Enter interactive mode. If specified, the `sfpread` utility emulates the front panel program and allows the front panel switches to be modified on a running system.
`-v`	Enter verbose mode. If specified in addition to a *field_name*, the `sfpread` utility prints the specified field name and its value in addition to returning that value to the shell.
field_name	If specified (with or without the `-v` option), a value corresponding to the current setting for that field is returned to the shell.

If the specified *field_name* does not exist or if any other error condition is found, the `sfpread` utility exits with exit status of -1. However, if invoked with the `-v` option, the `sfpread` utility always exits with status zero.

Files

/mnt/bin/sfpread

sp2util(1d)

Service processor register and memory utility

Syntax

```
sp2util
```

Function

Displays and modifies SPU memory locations. When the modify mode is entered, the displayed value can be changed by entering the new hex value and pressing **RETURN**. If no value is entered, the next memory location is displayed. A `q` terminates the modify and returns to the `Cmd:` prompt.

Commands

Commands are of the general form:

```
command parameters
```

All address values are in hex.

`q[uit]`	Terminates any command and returns to the `Cmd:` prompt. or exits the sp2util utility.
`?`	Prints out a list of valid commands.
`!`*command*	Executes the specified command. Use `!sh` to fork a shell.
`f` *a1* [*a2*] *value* `fb` *a1* [*a2*] *value*	The contents of memory locations *a1* through *a2* are filled with *value*, one byte at a time. If *a2* is not specified, only location *a1* is changed.
`fw` *a1* [*a2*] *value*	The contents of memory locations *a1* through *a2* are filled with *value*, one word at a time. If *a2* is not specified, only location *a1* is changed.
`fl` *a1* [*a2*] *value*	The contents of memory locations *a1* through *a2* are filled with *value*, one longword at a time. If *a2* is not specified, only location *a1* is changed.

`m` *a1* [*a2*] | `mb` *a1* [*a2*]	Displays the contents of memory locations *a1* through *a2*. If *a2* is not entered, location *a1* is displayed allowing modification of its contents. The locations are displayed and modified one byte at a time.
`mw` *a1* [*a2*]	Displays the contents of memory locations *a1* through *a2*. If *a2* is not entered, location *a1* is displayed allowing modification of its contents. The locations are displayed and modified one word at a time.
`ml` *a1* [*a2*]	Displays the contents of memory locations *a1* through *a2*. If *a2* is not entered, location *a1* is displayed allowing modification of its contents. The locations are displayed and modified one longword at a time.
`M` *a1 disp*	Causes the memory modify mode to be entered. Memory is displayed starting at *a1* with the option of modifying each byte displayed. The byte increment between displayed memory locations is *disp*.
`cp` *a1 a2 a3*	Copies a block of memory starting at *a1* through *a2* and places it at *a3*.
`tg` *a1 cnt val*	Writes *val* to *a1*, then waits for a period of time proportional to *cnt*; then writes out the `exclusive or` of *val* to *a1*. This command continues until interrupted by **CTRL-C** or **CTRL-?**.

`rm` *regname* Allows examination or modification of the hex contents of the named register. The contents of the register are displayed. The contents of the register can be modified by entering a hex value, if it is a writable register.

The following options are available to update and display registers:

RETURN	Updates the register and displays the next register.
CTRL	Updates the register and displays the previous register.
=	Updates the register and displays the same register again.
`q`	Exits this mode.

`rd` *regname* Allows examination or modification of the bits in the named register. The bits of the register are displayed. Each bit position is labeled with the function of the bit. Each bit can be individually modified by moving the cursor over the bit and entering 0 or 1.

The following options are available to move the cursor and to update and display registers:

SPACE BAR	The cursor moves to the right (towards the LSB) . Also used when a new bit value is entered.
DEL | BACK SPACE	The cursor moves to the left (towards the MSB) .
RETURN	Updates the register and displays the next register.
CTRL	Updates the register and displays the previous register.
=	Updates the register and displays the same register again.
`q`	Exits this mode.

The previous two commands modify memory-mapped, SPU-based registers. The valid register names are listed in Table 57.

Table 57
Register names
(C3400 Series)

Register Name	Description
ccsr	Console control/status register
cudr	Console data register
rcsr	Remote control/status register
rudr	Remote data register
tcsr	Timer control register
tdr	Timer data register
scsi_ctrl	SCSI status register
scsi_data	SCSI data register
ier_msw	Interrupt enable register (msw)
ier_lsw	Interrupt enable register (lsw)
isr_msw	Interrupt status register (msw)
isr_lsw	Interrupt status register (lsw)
lmcr	Local memory control register
cop	COP register
irs	SIB interrupt status register
icr	SIB interrupt channel register
ssn	System serial number
bsr	Bus error register
emr	Environmental monitor register
cpr	Control panel register
trr	Test result register
pcr	Physical configuration register
cfr	Clock frequency register
srr	System reset register
rfcnt	Refresh count register

Table 57 (continued) Register names (C3400 Series)

Register Name	Description
esr	Error source register
sel	Soft error log register
ebus_log	EBUS log register
rhr	Run/halt register
dcon	Diagnostic connect register
sfr	Scan feedback register
dcr	Diagnostic control register
sdr	Scan data register
early_clk	Early_clock register
ecr	Event counter register
odena	Output data enable register
mem_log	Memory log run register
mem	Memory run register
proc_a	Processor a run register
proc_b	Processor b run register
proc_c	Processor c run register
proc_d	Processor d run register
i/o	I/O run register
misc_log	Miscellaneous log run register
pbus_x	PBUS x clock mask register
pbus_y	PBUS y clock mask register
hi_level	Hi_level tester register

sram(1d)
Dump the scratch ram

Syntax

`sram [-c #] [-m | -r] [n1 [n2]]`

Function

Halts the specified CPUs and dumps the contents of the internal PTE cache (to stdout) as either RAM addresses or a logical addresses, in hexadecimal format.

The default operation of the `sram` utility is to dump the contents of the entire scratch ram for all the CPUs in the current configuration in physical address mode. The physical addresses are in the range `0x0` to `0xFFF`.

If an error is detected, the `sram` utility returns a -1. Otherwise, it returns a 0. In RAM address mode, an error is reported if an address is larger than is supported in the hardware.

Options

The following options are available:

`-c #`	Specifies a CPU (#) or all the CPUs in the current configuration. When you select a CPU number, the sram utility verifies that CPU is in the current configuration before attempting any dump operations. Default: All CPUs in the current configuration.
`-m`	Specifies memory address mode (default). Addresses *n1* through *n2* are logical addresses.
`-r`	Specifies RAM address mode. Addresses *n1* through *n2* are physical or RAM addresses.
`[n1 [n2]]`	Dumps addresses *n1* through *n2* of the scratch RAM. If no addresses are specified, then the contents of the entire scratch RAM are dumped.

Note

This utility applies only to C3400 Series computers.

See also

dcache(1d)
icache(1d)
ipte_cache(1d)
pte_cache(1d)

sos(1d)

Field service script to execute diagnostics from a data base of scripts and diagnostics

Syntax

sos [-fl #] *script_name* | -r | [-s | -S] *statement*

Function

The field service script utility, `sos`, resides on the SPU disk in the /mnt/sos directory. It allows a group of diagnostics (*script_name*) to be executed. It can also be used to execute selected single diagnostics (*statement*) from a script. The scripts can be viewed for the syntax of valid single `sos` statements (one per line as found in the scripts).

It is advisable to execute the diagnostic initialization utility prior to executing the field service scripts. Enter:

```
.diaginit -f
initall
```

The scripts listed in Table 58 are designed to functionally test each subsection of the system. They are available in the /mnt/sos/scripts directory.

Table 58
Available script names

Script name	Subsections tested
CPU	Scalars and vectors
CPU-QUICK	Scalars and vectors
INSTALL	All
IO	I/O
MEMORY	Memory (MCMs)

The `INSTALL` script should be executed at installation time. At least one of the above scripts should be executed prior to returning a failed board from the field.

When returning a printed circuit board to CONVEX Manufacturing, the `sos` utility generated report should also be returned. The report generated is /mnt/sos/logdir/report. It provides useful information to CONVEX Product Engineering.

Options

The following options are available (must be in the /mnt/sos directory):

script_name	Script to execute.
`-fl #`	Sets the fail limit (number of failures before stopping `sos` equals # + 1)) when executing a script.
`-r`	Resumes a script from the point it failed or was aborted.
-s *statement*	Executes a statement (diagnostic command with options). Does not initialize main memory.
-S *statement*	Executes a statement (diagnostic command with options). Initializes main memory.

To abort a script, type **CTRL-C**.

Operation

To run an `sos` script enter:

```
cd /mnt/sos
sos [-fl #] script_name
```

The selected script begins execution and checks for failures.
If `-fl` is used, # (a decimal number of) errors/failures are ignored.

To run a single `sos` statement from a script enter:

```
cd /mnt/sos
sos [-s | -S] statement
```

The selected statement begins execution and checks for failures.

To resume an `sos` script enter:

```
sos -r
```

The selected script resumes execution at the statement where it last failed (must be in the /mnt/sos directory).

Upon test completion, either a `We Failed` or `We Passed` message is displayed with the output written to /mnt/sos/logdir/report.

Example

The following examples run `sos` scripts:

```
sos INSTALL
```

This runs the diagnostic statements in the `INSTALL` script.

```
sos -fl 9 CPU
```

This runs the diagnostics in the `CPU` script until 10 failures occur or the script completes.

```
sos -s cpu4030 ff ipf
```

This runs `cpu4030` with force faults and IP force fault options on; all other options default. It does not initialize memory.

Notes

There may be variations in the time required to execute a script, as certain diagnostic tests within a script may be skipped if the system is not configured to execute them. The amount of memory in a system also affects the script execution time.

A hard copy of the report is obtained as follows:

1. Set the console printer on-line.
2. Enter **CTRL-ENTER** or **CTRL-PF4** at the console.
3. Enter `cat report` (in /mnt/sos/logdir directory).

`sos` margins the system to nominal all and ignores any margining done manually before executing the `sos` utility.

`sos` scans the output of the diagnostic it is running to determine if it passed of failed. The key word DIAG_BROKEN is printed to the report files if the diagnostic fails to output the correct message. This can be caused by a SPU hardware or software problem or a problem with `sos`.

Intermittent DIAG_BROKEN messages are being experienced in house. If this message occurs in the field, re-run the diagnostic under the dshell. If it then passes, ignore the DIAG_BROKEN message.

Files

/mnt/sos/scripts
/mnt/sos/report
/mnt/sos/times

sysreset(1d)
Reset the CPU(s)

Syntax

`sysreset` [[*subsystem* [*subsystem*]*] `-l` *level*]

Function

Resets a CPU or the CPU complex based on subsystems and reset levels.

The reset levels are subsystem dependent. Some subsystems have multiple levels of resetting, while others have a single default level. Attempting to reset a subsystem to a level higher than defined causes that subsystem to be reset to its maximum level.

Options

The following options are available:

subsystem — If no parameters are included, then all subsystem clocks are reset. Table 59 lists the valid subsystem names.

Table 59 Subystem identifiers

Subsystem	Description
cpus	All CPUs installed
mem	Memory subsystem
io	CPX

`-l` *level* — Set the reset level to *level*, an integer value defining what type of reset is being performed on the particular subsystem.

Note

Stopping subsystem clocks and stabilizing the hardware is a parallel operation, while resetting subsystems is a sequential operation.

Examples

The sysreset command can be used in five different ways to reset the indicated subsystems:

Example 1:

```
sysreset
```

Resets all subsystems to default level 0.

Example 2:

```
sysreset -l3
```

Resets all subsystems to default level 3.

Example 3:

```
sysreset cpus -l2
```

Resets processors to level 2.

Example 4:

```
sysreset cpus io -i1 mem -l3
```

Resets processors and IO to level 1 and the memory subsystem to level 3.

Example 5:

```
sysreset io
```

Resets the I/O subsystem to default level 0.

See also

reset(3d)
sys_init(3d)

version(1d)

Set/display parameters of diagnostic executables

Syntax

`version` [[`-V`] | [`-v` *version*]] *file* [*file* ...]

Function

Sets or displays version numbers in b.out and SOFF files, with magic numbers (octal) 405, 407, 410, 411, 425, 427, 430, 431, 601, and 603. Version numbers are of the form `X.X.X.X`, where `X` is an integer from 0 to 255. The time and date of compilation are displayed in the format returned by the ctime (3) function.

Information for each file (file name, file type, compile time and date, and version) is displayed on a separate line, as available. Displayed versions have at least the first two numbers. The last two numbers are displayed only if they are not zero.

Options

The following options are available:

The `-V` and `-v` options may be used to set the version information of the indicated file(s). When used, the option must be the first argument.

`-v` *version* — Increments a version number contained in the file /diag/bin/version.data, and provides a unique, incremented version number. It requires the version to be specified.

`-V` — Increments a version number contained in the file /diag/bin/version.data, and provides a unique, incremented version number. Regular use of the `-V` option allows easy tracking of unreleased software.

Both options are available only from the host executable. The service processor executable does not have the capability to set the version. The compilation time and date information contained in the header of SOFF format files is unaffected when setting the version. However, this information is set to the current time and date for b.out format files.

file — While the host executable allows only file names to be specified, either files or directories may be specified for the service processor executable.

In SOFF format files (magic numbers (octal) 601 and 603), the version number and compilation date fields are reserved in the file header (refer to a.out(5)).

version(1d)

In b.out format files (magic numbers (octal) 405, 407, 410, 411, 425, 427, 430, 431), the version number and compilation date are placed at the extreme end of the b.out format file (after the data relocation information).

Messages

The following messages can be output from this utility:

`file name is a directory - checking its files`

A directory is specified, and the service processor version prints the version of all SOFF and b.out files it contains.

`unable to open file name - skipping`

A file or directory cannot be opened.

`unable to fseek - file filename`

A problem is encountered moving around (`fseek`) in a file.

`unable to read file name - skipping`

A read of a file fails.

`unable to write [version | time] to [SOFF | b.out] file <filename>`

A write of version information to a file fails.

`version setting option available on host only`

If the set version option is invoked on the service processor.

`unable to process file name`

A file is not SOFF, b.out, or a service processor directory.

`no revision information present`

A b.out file contains no revision information.

`revision uninitialized`

A b.out file uninitialized revision information.

See also

a.out(5)
b.out(5)

x(1d)
Hexadecimal or decimal calculator

Syntax

`x [expression]`

Function

Evaluates an arithmetic expression using 32-bit integer arithmetic. The expression may include hexadecimal numbers, decimal numbers, octal numbers, zeros, and the operations - (unary negation), ~ (unary inversion), *, /, % (remainder), +, and -.

Unary operations are given the highest precedence; multiplication and division next; addition and subtraction have the lowest precedence. Parentheses may be used to override precedence.

Hexadecimal numbers are specified with leading zeros (example: `017`) or with a leading `x` (example: `x17`). Octal numbers are specified with a leading `o` (example: `o17`).

If no arguments are given, the `x` utility prompts for expressions, one per line.

To exit, respond with `q` or **RETURN**.

Diagnostic file formats

6

This chapter contains detailed explanations of the following diagnostic file formats:

- DB_cop
- Softlog

DB_cop(5d)
System board configuration database file

Description

DB_cop is an ASCII file that contains information on CONVEX system boards. Board entries are organized by CONVEX part numbers. Each part number is followed by the board type (such as mcm). In addition, one or more lines specify an assembly revision range with the corresponding diagnostic scan ring revision. Typically, DB_cop is used to translate the board part number and assembly revision stored in the board's COP chip to the scan ring revision used by various diagnostic tests.

Each line is either a comment, which starts with a "#", or is a board entry. An example format is illustrated in Figure 66.

Figure 66
Example DB_cop file

```
#  Cop Data Base File
#
#  Entry format:
#
#  PN       type
#           arr     rrn
#
#  where:   PN  = board part number (leading zeroes not needed)
#           type= type of board (mcm, asp, etc)
#           arr = assembly revision range (e.g., a, a-b)
#           rrn = scan ring revision number (1, 2, 3,...)
#
001201       cpx
             a-zz  1
002201       cpx
             a-zz  2
001205       vpc
             a-zz  1
002212       pia
             a-zz  2
```

See also

cop(1d)

softlog(5d)
Soft memory error log file for errintd

Description

This file contains a log of corrected main memory system single-bit errors, memory read access errors corrected by the error detection code (EDC). The `errintd` utility generates the softlog file, which consists of a header and a series of lines. One line per memory device (chip) is used. The header portion of the file contains the following items:

- The date the current softlog was created
- A field indicating whether the softlog is full
- A total error count
- An incremental count of failed devices (for example, number of softlog entries)
- An incremental count of errors not logged due to throttling by errintd

Each entry in the body of the softlog follows this format:

```
MCM device S/N bit stk first_fail last_fail
address count
```

`MCM` — Either the number of the memory board containing the faulty device (0-7) or, in later versions of `errintd`, a two character designation with a digit (0-3) indicating the board pair, and the letter (0 or e) indicating the individual odd or even board within the pair. For example, 2e = board pair 2, even board.

`device` — The coordinates of the faulty device, coded in either of the two following ways:

For the board MCM1 (P/N 1213), Table 60 lists the fields of the following format:

```
PP-SCCCRR
```

Table 60
MCM1 error soft log format

Code	Description
PP	Platter (examples: UL, UR, LL, LR)
S	Platter side (examples: U, Z)
CCC	Failed device column number (numeric characters only)
RR	Failed device row number (examples: A4, K5)

For the board MCM2 (P/N 3213), Table 61 lists the fields of the following format:

```
UKKKWW_UNN
```

Table 61
MCM2 error soft log format

Code	Description
U	Letter U
KKK	Column number of the MIM containing the failed RAM
WW	Row number of the MIM containing the failed RAM
NN	U-number of the failed RAM

`S/N` The serial number of the MCM containing the above device.

`bit` The bit that required correction. This only applies to the last corrected error that occurred on this device.

`stk` Indicates the repeatability of the error. The letter "S" indicates that although the error is correctable, it could not be eliminated by 10 attempted memory scrub operations of the address under test (that is, a bit is stuck at the address under test). This only applies to the last corrected error that occurred on this device.

`first-fail` The date the first error on this device occurred.

`last-fail` The date the most recent error on this device occurred.

`address` The address of the last single-bit error on this device. This only applies to the last corrected error that occurred on this device.

`count` The total number of single-bit errors (from this device) that have been corrected.

The softlog file may be examined from ConvexOS by entering the following command:

```
/usr/convex/spu -r /mnt/softlog
```

See also

errintd(1d)
mm_sniff(1d)

Diagnostic shell (dshell) 7

The diagnostic shell (dshell) is a command interface program that runs on the service processor (SP5) on the CONVEX C3400 Series CPUs. Most diagnostic tests are executed under control of the dshell.

The dshell has two basic functions:

- Presetting test options:
  - Pause on a failure or at the beginning or end of any specific subtest
  - Loop on a specific subtest or on a given set of subtests
  - Select subtest execution order
  - Direct test output to a file or to the screen (or both) to monitor the test as it runs or to analyze test results later
  - Select long or short error messages, or turn messages off
  - Execute user-created command scripts.
- Selecting diagnostics for execution

This chapter describes the dshell procedures for executing tests in manual mode and provides the basic information needed to execute diagnostics via the dshell. It describes the various options available for logging error and result messages, controlling test repetition, and others.

This chapter is particularly helpful to field service and manufacturing personnel who are responsible for board or system verification, or board or system debugging.

Invoking the dshell

To access the dshell, enter **dshell** at the spu > prompt. All dshell commands are then entered at the dshell prompt (:).

The sequencing and parameters for diagnostic testing can be determined either directly or indirectly via the dshell. Selection of the conditions under which subtests are executed is allowed. Low-level pass or fail data is returned.

dshell script files

Figure 67 typifies a dshell script file that will execute CONVEX CPU instruction set diagnostics.

Figure 67
Example dshell script file

```
pause off
pause -f
log off
log -t 999
msgs -f long
msgs -t
msgs -s
test cpu4030 -s 10-499,800-919 <T_ring0 +>cpu4030.0
test cpu4030 -s 10-499,800-919 <T_fault_icache +>cpu4030.0
```

In this example:

- The `pause off` command disables all pauses that were enabled before script execution.
- The `pause -f` command enables the "pause on fail" capability. If a failure occurs, test execution halts, and the user is prompted, enabling a service person to pursue the cause of the failure.
- The `log off` command disables previously enabled multiple failure logging.
- The `log -t 999` command instructs the dshell to allow multiple subtest failures per test without terminating the test.
- The `msgs` commands set up long-failure, test, and subtest messages, respectively.
- Two `test` commands are executed:
  - Both cause subtests numbered 10-499 and 800-919 to be executed.
  - Test command input is taken from two script files: `T_ring0` and `T_fault_icache`.
  - Test results are output to both the console and to the file cpu4030.0. The +> option appends results to the end of the file, leaving previous contents of the file undisturbed.

If this command file is named D_novec, and is present in /mnt/test on the service processor disk, simply invoke the dshell from SPU OS, and enter:

```
:<D_novec
```

This entry invokes the execution of all the commands contained in the script file D_novec.

dshell commands

The following pages explain the commands available at the dshell prompt (:).

access (!)
Access an operating system command

Description

Access, or `fork` to the ConvexOS shell, to execute the command that follows `!`.

Syntax

The standard form of this command is:

```
! [ OS_command ]
```

Operation

Any single command available under SPU OS can follow the `!`. Once the command has executed, program control reverts to the dshell.

Note

A test in progress has no knowledge of `access` commands occurring. If the machine state is destroyed while forked to a command, it is gone. A `pause` command can prevent this.

exit
Quit the dshell

Description

Exits from the diagnostic tests back to the dshell command level, or from the dshell to the SPU OS command level.

Syntax

There are four different forms of exit commands available in the dshell.

`e[xit]`
`q[uit]` — Immediately terminates the dshell process and any test processes that may have been forked.

CTRL-C — Returns you to the dshell command level, if no subtest is running. If subtest execution has started, the menu shown in Figure 68 appears.

Figure 68
dshell exit submenu

```
0 continue test
1 abort current test
2 abort current subtest
3 abort and pause at end of current subtest
```

CTRL- — Immediately terminates the dshell and any associated active processes. Core is dumped.

Operation

Test programs that interface with sensitive portions of the hardware protect certain portions of code from interrupts. This protection is recognized by the `exit`, `quit`, and **CTRL-C** commands. In addition, some test programs execute a clean-up routine before terminating, which leaves the hardware in a known state. The `exit`, `quit`, and **CTRL-C** commands allow this clean-up routine to execute.

The **CTRL-** command, however, is a dirty, nonmaskable exit that pays no heed to what is currently being executed. Sensitive code is not protected from **CTRL-**. In addition, **CTRL-** never allows the clean-up routine to execute. As a result, the hardware may, and probably will, be left in an indeterminate state if this command is used. Use **CTRL-** only as a last resort. System initialization should immediately follow.

Note

Using **CTRL-C** after a test command has been entered, but before the service processor has successfully forked the test process, may not kill the test process.

help
Online user aid

Description

Displays a standard help menu which describes the correct command syntax for each dshell command and gives a terse description of what each command does.

Syntax

The standard form of this command is:

```
help
```

Operation

The syntax for each dshell command is obtained by entering the command with no options.

Example

By entering `loop` with no options, the syntax help for the loop command is displayed on the screen, as shown in Figure 69.

Figure 69
Syntax help for the `loop` command

```
: loop
Proper syntax is:
loop off (-s) (-t)            :disables loop modes
loop -s nnn                   :loop on subtest nnn
loop -t                       :loop on test
```

status

Current state of the dshell command options

Description

Generates a report on the current state of the dshell command options which gives the name of each flag, its current value, and an explanation of its current effect.

Syntax

The standard form of the `status` command is:

```
status
```

Operation

The dshell keeps the current flag state in an external working file, named testflags, that is located in the current working directory. This file is deleted by the dshell on all exits except when you use **CTRL-**.

log
Record failures

Description

Specifies the number of failures allowed before a test or subtest terminates execution. Normally, dshell diagnostics terminate after a single failure.

Syntax

The standard form of the `log` command is:

```
log [options]
```

Options

The following options are available:

`off`	Disables all multiple failure logging
`off [-s] [-t]`	Disables previous log entries at the subtest or test level and terminates a test after the first failure
`-s` [*nn*]	Allows the tests to run until *nn* subtest failures have been logged
`-t` [*nn*]	Allows subtests to run until *nn* failures have been logged

The default setting is `off`.

Options `-s` and `-t`, with a subtest number, enable the command to execute at either the subtest or test level, if you want logging of a specific subtest.

Note

The `-s` flag does not work for all dshell tests. Check the appropriate chapter to determine whether the `-s` flag of a particular test is allowed.

loop
Repeat test execution

Description

Causes the dshell to repeat the execution of a test or subtest.

Syntax

The standard form of the command is:

```
loop [options]
```

Operation

Tests or subtests continue executing until the loop mode is disabled. Test looping terminates when **CTRL-C** is pressed.

To continue test execution after looping on a subtest, enter a `pause` command at the dshell prompt or in the script before the test execution commands. This allows for disabling subtest looping when desired (`loop off -s`) and continuing normal test execution.

If test looping is enabled, the dshell records the pass or fail result of each test iteration. When **CTRL-C** is pressed, and test looping terminates, the results are summarized and written to the selected output.

Options

The following options are available:

`off`	Disables all looping
`off [-s] [-t]`	Disables looping at the subtest or test level
`-s`	Allows looping on every subtest
`-s [`*nn*`]`	Allows looping on subtest *nn*, if it is executed
`-t`	Allows looping on the entire test

The default setting is `off -s -t`.

msgs
Record messages

Description

Enables or disables different levels of test, class, and subtest result messages.

Syntax

The basic format of the `msgs` command is:

```
msgs [options]
```

Options

The following options are available:

`off [-f] [-s] [-t]`	Disables all or specific test messages
`-f [long | short]`	Enables long or short failure messages
`-s`	Enables subtest result messages
`-t`	Enables test result messages

The default setting is `msgs -f long -s -t`.

pause
Interrupt testing

Description

Returns program control to the dshell:

- At the beginning of all or specific subtests
- When a failure is encountered in all or specific subtests
- At the end of all or specific subtests.

Syntax

The standard form of the `pause` command is:

```
pause [option] [nn]
```

where *nn* signifies the number of a specific subtest.

Each of the variations of `pause` can be set up either for all subtests or for a particular subtest.

Operation

When a pause occurs, program control is returned to the dshell command level. Any SPU OS command or diagnostic utility can then be executed by using the `access` command (`!`).

When failure analysis or SPU OS command execution has been completed, pressing **RETURN** causes the test sequence to be resumed.

Options

The following options are available:

`off`	Disables all pauses
`off [-f] [-b] [-e]`	Disables specified pauses
`-f`	Enables a pause if a failure is encountered during execution of any subtest
`-f [nn]`	Enables a pause if a failure is encountered during execution of subtest *nn*
`-b`	Enables a pause at the beginning of each subtest that is executed
`-b [nn]`	Enables a pause at the beginning of subtest *nn*, if it is executed
`-e`	Enables a pause at the end of each subtest that is executed

`-e` [*nn*]	Enables a pause at the end of subtest *nn*, if it is executed. Omitting the argument *nn* causes a pause to be enabled for each subtest (where applicable).

The default setting is `pause off`.

Notes

The test has no knowledge of `access` commands occurring. If the machine state is destroyed while forked to a command, it is gone.

If an `-e` pause is enabled and a failure occurs, it is possible that the pause will not be taken. This happens when logging of multiple failures per test (`log -t`) is not enabled, or when the current failure is the last failure permitted by the count entered with a `log -t` command.

test
Execute a diagnostic test

Description

Executes specific tests, and displays test, class, and subtest menus.

Syntax

To execute a test, at the dshell prompt or in a script file, enter

```
test [testname] [options]
```

where *testname* is the test to be executed.

To access a menu only at the dshell prompt, enter

```
test
```

This command displays a list of all tests available within the current working directory, as well as the tests located in /mnt/test. Following the list, a menu is displayed as illustrated in Figure 70.

Figure 70
dshell working directory menu

```
<CR>  Display next screen of test menu
p     Display previous screen of test menu.
t     Display top page of test menu.
b     Display bottom page of test menu.
?     Display help menu.
q     Leave the menu.
```

Using this form, either a specific test can be executed or additional subtest menus can be displayed. To display a specific subtest menu, enter

```
test [testname] [option]
```

where *testname* is the test to be executed or displayed.

The `-s` option causes the subtest menu to be displayed.
The `-c` option causes the class menu to be displayed.

Options

The following options are available:

no option	Presents a menu listing all tests available under the dshell
[*testname*]	Executes all subtests for *testname*
[*testname*] -c	Presents the class menu for *testname*
[*testname*] -s	Presents the subtest menu for *testname*
[*testname*] -s [*subtest list*]	Executes listed subtests in the order specified
[*testname*] -c [*class list*]	Executes all subtests within the listed classes in the order specified

Subtest and class specification syntax is identical.

Default test execution is defined to be execution of each subtest that is available in a test, once, in ascending order.

Examples

When entering a test command while specifying the subtest or class list, it is also possible to arrange the execution order of the subtests or classes specified and to execute multiple iterations of specific subtests or classes, or groups of subtests or classes. This is illustrated by the following examples:

Example 1:

```
-s 1
```

Execute subtest 1.

Example 2:

```
-s 1, 5
```

Execute subtests 1, 5.

Example 3:

```
-s 1-5
```

Execute subtests 1 through 5.

Example 4:

```
-s 2(1-5)
```

Execute subtests 1 through 5, two times.

Example 5:

```
-s 1, 5, 3(5, 4)
```

Execute subtests 1, 5, 5, 4, 5, 4, 5, 4.

Notes

To use specific dshell commands (`log`, `loop`, `pause`, and `msgs`), enter those commands before entering a command for the test to execute. Alternatively, create a script file with the commands and enter the name of the script file.

For example, to run mem4100 subtests until three failures have been logged with no pauses, enter:

```
log -s 3
pause off
test mem4100
```

This causes `mem4100` to begin execution and run until three failures occur or until the test completes.

To redirect input and output, use the following notational conventions:

< *filename*	Causes input to be taken from the specified file
> *filename*	Causes standard output to be appended to the specified file, but not to the screen
+> *filename*	Causes standard output to be appended to both the specified file, and to the screen

< *filename*, > *filename*, and +> *filename* may be appended to the end of any form of the `test` command. However, > and +> are mutually exclusive.

Service processor interface tests (spu4000)

8

The service processor interface test (spu4000) verifies the interface between the service processor and various field replaceable units (FRUs).

This test is designed to verify the hardware in a hierarchical fashion. First, control logic contained on the service processor is checked to ensure it is working correctly, then initial testing of the interfaces between the service processor and other parts of the system occurs. This is followed by more extensive testing of the service processor to system interfaces.

This test should be run only after the EPROM-based self-test has passed.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 62, which represents the minimum configuration for running this test. Additional field replaceable units (FRUs) installed in the system may be tested, depending on the response(s) made in the configuration menu.

Table 62
`spu4000` functional areas tested

Functional area	Tested by diagnostic	Exercised by diagnostic
CPU	No[1]	No
CUJ	No[1]	Yes
Memory even	No[1]	No
Memory odd	No[1]	No
PI2	No[1]	Yes
CCU	No[1]	No
SP5	Yes	No

[1] Tests the scan ring integrity on any board specified in the configuration menu. Tests the COP chip on all boards.

In general, each subtest requires only the service processor, the CUJ, and the FRU which generates clocks, and possibly a FRU to be tested. The "Probable fault location" column of each subtest table lists the minimum FRUs required to run each subtest.

In addition:

- Subtest 1200 requires at least one MCM be installed in any memory slot.
- Subtest 7200 requires both memory odd 0 (MO0) and memory even 0 (ME0).
- Subtest 8010 requires the CUJ.

Many `spu4000` subtests check one of several interfaces between the service processor and another FRU, and require the FRU to be installed to run the subtest. To provide for a wide variety of testable configurations, a method of specifying FRUs to be tested has been provided. The "Configuration menu" section in this chapter explains the method of FRU specification.

Only subtests for FRUs specified in the configuration menu are attempted. Subtests which require FRUs not contained in the test configuration are not executed.

Test invocation

To invoke the spu4000 test, use the procedure shown in Figure 71.

Figure 71
spu4000 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL

: test spu4000 [-c [class...]] [-s [subtest...]] [+> filename]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to the "Diagnostic shell (dshell)" chapter of this manual for more information.

Caution

If the system has just been powered up, if mem4100 was executed with failures, or if spu4000 was executed, then initall must be executed prior to test execution. It should be run only after the EPROM based self-test has passed. Failure to execute initall in these circumstances could result in invalid test results.

Entering only

```
test spu4000
```

executes all spu4000 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the `-c` or `-s` options, respectively. Detailed information for using these options can be found in the "Diagnostic shell (dshell)" chapter of this manual.

The [+>*filename*] option allows the test results to be appended to *filename*.

Configuration menu

When spu4000 is invoked, it first displays the current configuration, as illustrated in Figure 72. This is a list of FRUs it believes are installed in the system, determined by examining information generated by the scnlink utility. This list of FRUs should match the one in the file /mnt/user/scn/cop.out, which is generated using the utility cop.

Figure 72
spu4000 configuration dialog

```
Test 'spu4000.t'                              Tue Sep 15 00:00:00 1964

The current configuration includes the following slots:
  cpu0 cpu1 sp5 cuj me0 mo0 me1 mo1 piy ccu0 ccu1

[command [boards]] ... (end) :
> remove all

The current configuration includes the following slots:

[command [boards]] ... (end) :
> add mcm piy cuj sp5

The current configuration includes the following slots:
  sp5 cuj mo0 me0 mo1 me1 mo2 me2 mo3 me3 piy

[command [boards]] ... (end) :
> remove mcm1 mcm2 mcm3e

The current configuration includes the following slots:
  sp5 cuj mo0 me0 mo3 piy

[command [boards]] ... (end) :
> add ccu0 add ccu1 remove mcm3o end

The test configuration includes the following slots:
  sp5 cuj mo0 me0 piy ccu0 ccu1

[command [boards]] ... (end) :
> RETURN

      Subtest 1000  0:00:00 passed
      Subtest 1100  0:00:00 passed
            .
            .
            .
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

As Figure 72 shows, commands may be entered to alter the configuration only immediately following the prompt:

```
[command [boards]]... (end) :
>
```

If no attempt is made to change the configuration at this prompt (if you press only **RETURN**), testing occurs. Otherwise the configuration is modified as requested, and the prompt reissued, allowing you to further alter the configuration or begin testing.

Before performing any tests, the current configuration should be examined, and modified if necessary, since subtests which require hardware not in the specified configuration are not attempted. After any modifications have been made to the current configuration, the list of FRUs to be tested is displayed. This list is called the test configuration. Only FRUs contained in the test configuration will be tested.

Three commands are available to change the configuration. These commands may be used individually or in combination on an input line, with commands processed left to right. In addition, each command may be abbreviated by entering only the first letter.

a[dd] — Adds FRUs to the current configuration. The `add` command is illustrated in the previous figure where the mcm, pia, cpx, and sp2 terms are added to a configuration where no FRUs are specified. Adding these FRUs, as shown, returns a configuration which contains all eight MCMs as well as the pia, cpx and sp2. The `add` command may be followed by any number of terms.

r[emove] — Removes FRUs from the current configuration. The `remove` command is illustrated in the previous figure where the mcm1, mcm2, and mcm3 terms are removed from the configuration. Removing these terms, as shown, returns a configuration which no longer contains the mo1, me1, mo2, me2, and me3 FRUs. The `remove` command may be followed by any number of terms.

`end` Indicates testing should begin, and that no other input should be processed or requested. The `end` command is used in the previous figure at the end of a line which contains add ccu0, add ccu1, remove mcm3, end. This `end` command can be used on a blank line, or at the end of an input line containing one or more add or remove commands. The `end` command may also be indicated by pressing on a blank line. When the `end` command is entered, the machine prints the test configuration, and begins subtest execution.

Table 63, Table 64, and Table 65 indicate the possible terms which may be used to add or remove FRUs from the test configuration.

The current configuration contains only entries from the "Single slot terms" columns of the tables. The "Multiple slot terms" columns of the tables contain terms which represent multiple FRUs, or slots. You may enter either multiple-slot or single-slot terms for your convenience and `spu4000` will recognize them as input. However, `spu4000` lists only single-slot terms for the FRUs in a configuration.

To avoid ambiguity, each single-slot term can represent only one FRU. Note that the terms listed in the tables cannot all apply to any given system. Those terms which do not apply are ignored.

Table 63
CPU FRU configuration terms

Multiple slot terms	Description	Single slot terms	Description
all	All FRUs	cpu0	cpu0
cpu	All CPU FRUs	cpu1	cpu1
		cpu2	cpu2
		cpu3	cpu3
		cpu4	cpu4
		cpu5	cpu5
		cpu6	cpu6
		cpu7	cpu7

Table 64
Memory configuration terms

Multiple slot terms	Description	Single slot terms	Description
mcm	All memory FRUs	mo0	Memory 0 odd
mcme	All memory even FRUs	me0	Memory 0 even
mcmo	All memory odd FRUs	mo1	Memory 1 odd
mcm0	Both memory 0 FRUs	me1	Memory 1 even
mcm1	Both memory 1 FRUs	mo2	Memory 2 odd
mcm2	Both memory 2 FRUs	me2	Memory 2 even
mcm3	Both memory 3 FRUs	mo3	Memory 3 odd
		me3	Memory3 even

Table 65
I/O FRU configuration terms

Multiple slot terms	Description	Single slot terms	Description
io	All PI2, CU, and CCU FRUs	cuj	CUJ
cu	All CPU utility FRU(s)	piy	PIY
pi	All PBUS interface FRU(s)	pix	PIX
ccu	All CCU FRUs	ccu0	CCU 0, PBUS Y
ccuy	All PBUS Y CCU FRUs	ccu1	CCU 1, PBUS Y
ccux	All PBUS X CCU FRUs	ccu2	CCU 2, PBUS Y
		ccu4	CCU 4, PBUS X
		ccu5	CCU 5, PBUS X
		ccu6	CCU 6, PBUS X

Class descriptions

The service processor interface test is divided into nine classes, listed in Table 66.

The "Test performed" column describes the hardware tested.

Table 66
Service processor interface test classes

Class	Class name	Test performed
1	SPU registers	Verifies most of the service processor registers
2	DBUS interface integrity	Verifies the scan ring interface
3	Board ID test	Checks ability to read each board's COP T
4	DBUS integrity	Pattern tests scan rings on boards
5	Hard-error test	Forces generation, and checks reporting of hard errors
6	Soft-error test	Forces generation, and checks reporting of soft errors
7	EBUS interface	Checks SP5 interface with EBUS
8	Interrupt bus integrity	Tests system interrupt bus operation
9	Margin test	Tests ability to margin power supplies and clocks

Subtest descriptions

The following sections describe the subtests for each class, and explain the hardware function tested.

Each class section starts with a description of that class of subtests. Following this is a table which contains each subtest in the class. The "Probable fault location" column lists one or more field replaceable units (FRUs) required to execute the subtest (in addition to the FRU which generates system clocks). The failure of any of the listed FRUs could cause a subtest to fail, however, it is possible the problem that causes a subtest to fail may not be located on any of the FRUs indicated in the "Probable fault location column."

It is important to note that all subtests do not apply to all systems, and that only those subtests that apply to FRUs in the test configuration are attempted.

Class 1 subtests—Service processor registers

Subtests 1000-1300, listed in Table 67, test some of the service processor's registers, the system serial number, the run-arm circuitry, and the service processor's realtime clock.

Table 67
Class 1 subtests

Subtest	Test performed	Probable fault location
1000	SP5 control register pattern test	SP5
1100	System serial number validity	SP5
1200	SP5 run arm circuitry	SP5, MCM
1300	SP5 realtime clock	SP5

Subtest 1000—Service processor control register

Subtest 1000 writes various patterns to each of the registers listed in Table 68 (by acronyms). The pattern written to a register could be one of the following:

- Alternating ones and zeros (10101010)
- Alternating zeros and ones (01010101)

For each pattern and register, the register is written with the pattern. Next, the register is read, and then the data is checked against the pattern to ensure the register is functioning correctly.

Table 68
Registers tested by subtest 1000

Acronym	Register name
CPR	Control panel register
DCON	Diagnostic connect register
DCR	Diagnostic control register
IO	I/O run register
MEM	Memory run register
MEM_LOG	Memory log run register
MISC_LOG	I/O log run register
ODENA	Output data enable register
PBUS_X	PBUS X CCU run register
PBUS_Y	PBUS Y CCU run register
PCR	Physical configuration register
PORT_A	PORT A run register
PORT_B	PORT B run register
PORT_C	PORT C run register
PORT_D	PORT D run register
RHR	Run halt register
SRR	System reset register
TRR	Test result register

Subtest 1100—System serial number validity

Subtest 1100 tests the system serial number and ensures that the machine class is within an allowable range and the serial number is not all zeros or all ones.

Subtest 1200—Service processor run-arm circuitry

Subtest 1200 tests the service processor run-arm circuitry. This subtest requires that at least one MCM be installed in the system. The current state of the service processor is saved before testing starts, and restored after the test completes.

The following checks are made on the bits in the diagnostic control register, in this order:

- Busy bit clear before hitting go
- Run_arm bit clear before hitting go
- Ecr_not_zero bit set before hitting go
- Run_arm bit sets within a reasonable period of time
- Busy bit set after run_arm bit set
- Ecr_not_zero bit set after run_arm bit set
- Busy bit clears within a reasonable period of time
- Run_arm bit set when busy cleared
- Ecr_not_zero bit cleared when busy cleared

Subtest 1300—Service processor realtime clock

Subtest 1300 checks the real time clock over a fraction of a second to ensure it is operational.

Class 2 subtests—DBUS interface

Class 2 subtests verify the ability of the service processor to interface with various FRUs through the diagnostics bus (DBUS), which is used to access scan rings.

Subtest 2000—Service processor scan loopback

Subtest 2000 checks the service processor's scan interface to the DBUS via loopback. When in loopback mode, the data shifted out of one end of the scan data register is shifted into the other end of the register. A single set bit is rotated through the scan data register in both directions. Each of the 16 bits in the scan data register is individually tested, with the rotate count varied from 0 to 16, for each scan output data enable (odena).

Subtests 2108-2406—Service processor scan communicate

Subtests 2108 to 2406 are the first subtests which actually send data to, and receive data from, the FRUs listed in Table 69, Table 70, and Table 71, using scan.

These class 2 subtests test the scan ring interface between the system FRUs and the service processor by scanning the least significant Bits (LSB) and the most significant bits (MSB) of each scan ring with a 0 and a 1, if possible.

The following tests are performed in the indicated order for Class 2 subtests:

1. Fully bidirectional scan ring
   a. LSB, with a 0
   b. LSB, with a 1
   c. MSB, with a 0
   d. MSB, with a 1
2. Partly bidirectional scan rings
   a. LSB, with a 0
   b. LSB, with a 1
3. Fully unidirectional scan rings—neither the LSB nor the MSB is testable

Table 69
Class 2 service processor and CPU subtests

Subtest	Test performed	Probable fault location
2000	SPU scan loopback	SP5
2108 2109	CPU0 scan interface CPU1 scan interface	CPU0 CPU1
2118 2119	CPU2 scan interface CPU3 scan interface	CPU2 CPU3
2128 2129	CPU4 scan interface CPU5 scan interface	CPU4 CPU5
2138 2139	CPU6 scan interface CPU7 scan interface	CPU6 CPU7

Table 70
Class 2 memory FRU subtests

Subtest	Test performed	Probable fault location
2200	Memory 0 even scan interface	ME0, SP5
2202	Memory 0 even log scan interface	ME0, SP5
2205	Memory 0 odd scan interface	MO0, SP5
2207	Memory 0 odd log scan interface	MO0, SP5
2200	Memory 1 even scan interface	ME1, SP5
2202	Memory 1 even log scan interface	ME1, SP5
2205	Memory 1 odd scan interface	MO1, SP5
2207	Memory 1 odd log scan interface	MO1, SP5
2200	Memory 2 even scan interface	ME2, SP5
2202	Memory 2 even log scan interface	ME2, SP5
2205	Memory 2 odd scan interface	MO2, SP5
2207	Memory 2 odd log scan interface	MO2, SP5
2200	Memory 3 even scan interface	ME3, SP5
2202	Memory 3 even log scan interface	ME3, SP5
2205	Memory 3 odd scan interface	MO3, SP5
2207	Memory 3 odd log scan interface	MO3, SP5

Table 71
Class 2 I/O FRU subtests

Subtest	Test performed	Probable fault location
2335	CUJ scan interface	CUJ, SP5
2337	CUJ scan interface	CUJ, SP5
2350	PIY scan interface	PIY, SP5
2352	PIY scan interface	PIY, SP5
2354	PIY scan interface	PIY, SP5
2360	PIX scan interface	PIX, SP5
2362	PIX scan interface	PIX, SP5
2364	PIX scan interface	PIX, SP5
2400	CCU0 scan interface	CCU 0, SP5
2401	CCU1 scan interface	CCU 1, SP5
2402	CCU2 scan interface	CCU 2, SP5
2404	CCU4 scan interface	CCU 4, SP5
2405	CCU5 scan interface	CCU 5, SP5
2406	CCU6 scan interface	CCU 6, SP5

Class 3 subtests—Board ID verification

Class 3 subtests check the COP of a particular FRU. A COP is a nonvolatile RAM which contains information about a FRU, which may include the FRU type, revision level, and serial number. These subtests attempt to read the COP for each FRU in the configuration. If a COP read is successful, the subtest then checks to see if the type of FRU is known and if it is allowed in the backplane slot where it is found.

Subtests 3108-3139—CPU COP chip verification

Subtests 3108-3139 , listed in Table 72, test the COP of the CPU boards.

Table 72
Class 3 CPU FRU subtests

Subtest	Test performed	Probable fault location
3108 3109	CPU0 COP CPU1 COP	CPU0, SP5 CPU1, SP5
3118 3119	CPU2 COP CPU3 COP	CPU2, SP5 CPU3, SP5
3128 3129	CPU4 COP CPU5 COP	CPU4, SP5 CPU5, SP5
3138 3139	CPU6 COP CPU7 COP	CPU6, SP5 CPU7, SP5

Subtests 3200-3235—Memory COP chip verification

Subtests 3200-3235, listed in Table 73, test the COP of the memory boards.

Table 73
Class 3 Memory FRU subtests

Subtest	Test performed	Probable fault location
3200	Memory 0 even COP	ME0, SP5
3205	Memory 0 odd COP	MO0, SP5
3210	Memory 1 even COP	ME1, SP5
3215	Memory 1 odd COP	MO1, SP5
3220	Memory 2 even COP	ME2, SP5
3225	Memory 2 odd COP	MO2, SP5
3230	Memory 3 even COP	ME3, SP5
3235	Memory 3 odd COP	MO3, SP5

Subtests 3300-3406—Miscellaneous COP chip verification

Subtests 3300-3406 , listed in Table 74, test the COP of several miscellaneous boards.

Table 74
Class 3 I/O FRU subtests

Subtest	Test performed	Probable fault location
3300	SP COP	SP5
3335	CUJ COP	CUJ, SP5
3350	PIY COP	PIY, SP5
3360	PIX COP	PIX, SP5
3400	CCU0 COP	CCU0, SP5
3401	CCU1 COP	CCU1, SP5
3402	CCU2 COP	CCU2, SP5
3404	CCU4 COP	CCU4, SP5
3405	CCU5 COP	CCU5, SP5
3406	CCU6 COP	CCU6, SP5

Subtests 3708-3750—COP RAM chip verification

Subtests 3708-3750, listed in Table 75, verify that each COP chip tested can hold the particular data written to it. Each cop is tested with two address uniqueness patterns, and each location is verified to make sure all bits can be set and cleared.

Table 75
Class 3 COP RAM subtests

Subtest	Test performed	Probable fault location
3708	CPU0 COP RAM	CPU0, SP5
3709	CPU0 COP RAM	CPU0, SP5
3718	CPU0 COP RAM	CPU0, SP5
3719	CPU0 COP RAM	CPU0, SP5
3728	CPU0 COP RAM	CPU0, SP5
3729	CPU0 COP RAM	CPU0, SP5
3738	CPU0 COP RAM	CPU0, SP5
3739	CPU0 COP RAM	CPU0, SP5
3750	CUJ COPP RAM TEST	CUJ, SP5

Class 4 subtests—Scan ring integrity

Class 4 subtests test the scan rings on the system FRUs by scanning patterns through them in both directions, as possible. Each pattern consists of all bits either set or clear, except bit 0, which is inverted compared to the rest. If the data read from a scan ring does not match the pattern written to it, failure isolation may be attempted.

The following list describes the isolation attempted for each of the three types of scan rings:

- **Fully bidirectional**—Full isolation attempted for any failure.
- **Partly bidirectional**—Isolation attempted only when the initial failure is in the LEFT direction.
- **Fully unidirectional**—No isolation attempted (none possible).

If isolation is possible, it is attempted by scanning a number of bits into the ring in the same direction the failure was detected, then scanning the same number of bits out in the opposite direction. The number of bits scanned starts at one, and is incremented until either the failing bit is detected (the data read does not match the data written), or the shift count equals the length of the bidirectional part of the scan ring.

For any failure, an appropriate error message is printed, describing the failure and the results of any isolation attempts.

The following list shows the order of testing:

1. Any bidirectional portion of the ring under test is scanned in the LEFT direction, with all bits of the scan ring set, except bit zero.
2. Any bidirectional portion of the ring under test is scanned in the LEFT direction, with only bit zero set.
3. The ring under test is scanned in the RIGHT direction, with all bits of the scan ring set, except bit zero.
4. The ring under test is scanned in the RIGHT direction, with only bit zero set.

Subtests 4108-4139—CPU scan ring integrity

Subtests 4108-4139, listed in Table 76, verify the CPU FRU scan ring integrity.

Table 76
Class 4 CPU FRU subtests

Subtest	Test performed	Probable fault location
4108	CPU0 scan ring integrity	CPU0, SP5
4109	CPU1 scan ring integrity	CPU1, SP5
4118	CPU2 scan ring integrity	CPU2, SP5
4119	CPU3 scan ring integrity	CPU3, SP5
4128	CPU4 scan ring integrity	CPU4, SP5
4129	CPU5 scan ring integrity	CPU5, SP5
4138	CPU6 scan ring integrity	CPU6, SP5
4139	CPU7 scan ring integrity	CPU7, SP5

Subtests 4200-4207—Memory scan ring integrity

Subtests 4200-4207, listed in Table 77, verify the memory FRU scan ring integrity.

Table 77
Class 4 Memory FRU subtests

Subtest	Test performed	Probable fault location
4200	Memory 0 even scan ring integrity	ME0, SP5
4202	Memory 0 even log scan ring integrity	ME0, SP5
4205	Memory 0 odd scan ring integrity	MO0, SP5
4207	Memory 0 odd log scan ring integrity	MO0, SP5
4200	Memory 1 even scan ring integrity	ME1, SP5
4202	Memory 1 even log scan ring integrity	ME1, SP5
4205	Memory 1 odd scan ring integrity	MO1, SP5
4207	Memory 1 odd log scan ring integrity	MO1, SP5
4200	Memory 2 even scan ring integrity	ME2, SP5
4202	Memory 2 even log scan ring integrity	ME2, SP5
4205	Memory 2 odd scan ring integrity	MO2, SP5
4207	Memory 2 odd log scan ring integrity	MO2, SP5
4200	Memory 3 even scan ring integrity	ME3, SP5
4202	Memory 3 even log scan ring integrity	ME3, SP5
4205	Memory 3 odd scan ring integrity	MO3, SP5
4207	Memory 3 odd log scan ring integrity	MO3, SP5

Subtests 4335-4406—Miscellaneous scan ring integrity

Subtests 4335-4406, listed in Table 78, verify miscellaneous FRU scan ring integrity.

Table 78
Class 4 I/O FRU subtests

Subtest	Test performed	Probable fault location
4335	CUJ scan ring integrity	CPX, SP5
4337	CUJ log scan ring integrity	CPX, SP5
4350	PIY scan ring integrity	PIY, SP5
4352	PIY log scan ring integrity	PIY, SP5
4354	PIY auxiliary scan ring integrity	PIY, SP5
4360	PIX scan ring integrity	PIX, SP5
4362	PIX log scan ring integrity	PIX, SP5
4364	PIX auxiliary scan ring integrity	PIX, SP5
4400	CCU0 scan ring integrity	CCU0, SP5
4401	CCU1 scan ring integrity	CCU1, SP5
4402	CCU2 scan ring integrity	CCU2, SP5
4404	CCU4 scan ring integrity	CCU4, SP5
4405	CCU5 scan ring integrity	CCU5, SP5
4406	CCU6 scan ring integrity	CCU6, SP5

Subtests 4600-4617—STRAM scan ring functionality

Subtests 4600-4617, listed in Table 79, verify the STRAM scan ring functionality, by scanning patterns into the scan ring, then reading them back and verifying them. Since these rings scan in only one direction, no isolation is available in the event of a failure.

Table 79
Class 4 STRAM integrity subtests

Subtest	Test performed	Probable fault location
4600	CPU0 LEFT STRAM scan ring integrity	CPU0, SP5
4601	CPU1 LEFT STRAM scan ring integrity	CPU1, SP5
4602	CPU2 LEFT STRAM scan ring integrity	CPU2, SP5
4603	CPU3 LEFT STRAM scan ring integrity	CPU3, SP5
4604	CPU4 LEFT STRAM scan ring integrity	CPU4, SP5
4605	CPU5 LEFT STRAM scan ring integrity	CPU5, SP5
4606	CPU6 LEFT STRAM scan ring integrity	CPU6, SP5
4607	CPU7 LEFT STRAM scan ring integrity	CPU7, SP5
4610	CPU0 RIGHT STRAM scan ring integrity	CPU0, SP5
4611	CPU1 RIGHT STRAM scan ring integrity	CPU1, SP5
4612	CPU2 RIGHT STRAM scan ring integrity	CPU2, SP5
4613	CPU3 RIGHT STRAM scan ring integrity	CPU3, SP5
4614	CPU4 RIGHT STRAM scan ring integrity	CPU4, SP5
4615	CPU5 RIGHT STRAM scan ring integrity	CPU5, SP5
4616	CPU6 RIGHT STRAM scan ring integrity	CPU6, SP5
4617	CPU7 RIGHT STRAM scan ring integrity	CPU7, SP5

Subtests 4620-4647—Signature verification

Subtests 4620-4647, listed in Table 80, verify the signatures in either the gate arrays or the STRAMs. The board is clocked, then the scan ring is read, and the fields are checked to verify they contain the correct signature value.

Table 80
Class 4 signature subtests

Subtest	Test performed	Probable fault location
4620	CPU0 gate array signature	CPU0, SP5
4621	CPU1 gate array signature	CPU1, SP5
4622	CPU2 gate array signature	CPU2, SP5
4623	CPU3 gate array signature	CPU3, SP5
4624	CPU4 gate array signature	CPU4, SP5
4625	CPU5 gate array signature	CPU5, SP5
4626	CPU6 gate array signature	CPU6, SP5
4627	CPU7 gate array signature	CPU7, SP5
4630	CPU0 LEFT STRAM signature	CPU0, SP5
4631	CPU1 LEFT STRAM signature	CPU1, SP5
4632	CPU2 LEFT STRAM signature	CPU2, SP5
4633	CPU3LEFT STRAM signature	CPU3, SP5
4634	CPU4 LEFT STRAM signature	CPU4, SP5
4635	CPU5 LEFT STRAM signature	CPU5, SP5
4636	CPU6 LEFT STRAM signature	CPU6, SP5
4637	CPU7 LEFT STRAM signature	CPU7, SP5
4640	CPU0 RIGHT STRAM signature	CPU0, SP5
4641	CPU1 RIGHT STRAM signature	CPU1, SP5
4642	CPU2 RIGHT STRAM signature	CPU2, SP5
4643	CPU3 RIGHT STRAM signature	CPU3, SP5
4644	CPU4 RIGHT STRAM signature	CPU4, SP5
4645	CPU5 RIGHT STRAM signature	CPU5, SP5
4646	CPU6 RIGHT STRAM signature	CPU6, SP5
4647	CPU7 RIGHT STRAM signature	CPU7, SP5

Class 5 subtests—Hard error interrupts

Class 5 subtests verify that each FRU in the system capable of generating a hard error can cause the generation of a hard-error interrupt on the service processor, and that the service processor can correctly determine the functional unit sending the error. Scan operations are used to set and clear a hard error condition on the FRU under test.

Subtests 5108-5139—CPU hard error interrupt

Subtests 5108-5139, listed in Table 81, verify the CPU FRUs can generate hard error interrupts.

Table 81
Class 5 CPU FRU subtests

Subtest	Test performed	Probable fault location
5108 5109	CPU0 hard error CPU1 hard error	CPU0, SP5 CPU1, SP5
5118 5119	CPU2 hard error CPU3 hard error	CPU2, SP5 CPU3, SP5
5128 5129	CPU4 hard error CPU5 hard error	CPU4, SP5 CPU5, SP5
5138 5139	CPU6 hard error CPU7 hard error	CPU6, SP5 CPU7, SP5

Subtests 5200-5235—Memory hard error interrupt

Subtests 5200-5235, listed in Table 82, verify the memory FRUs can generate hard error interrupts.

Table 82
Class 5 memory subtests

Subtest	Test performed	Probable fault location
5200	Memory 0 even hard error	ME0, SP5
5205	Memory 0 odd hard error	MO0, SP5
5210	Memory 1 even hard error	ME1, SP5
5215	Memory 1 odd hard error	MO1, SP5
5220	Memory 2 even hard error	ME2, SP5
5225	Memory 2 odd hard error	MO2, SP5
5230	Memory 3 even hard error	ME3, SP5
5235	Memory 3 odd hard error	MO3, SP5

Subtests 5335-5360—Miscellaneous hard error interrupts

Subtests 5335-5360, listed in Table 83, verify miscellaneous FRUs can generate hard error interrupts.

Table 83
Class 5 I/O FRU subtests

Subtest	Test performed	Probable fault location
5335	CUJ hard error	CUJ, SP5
5350	PIY hard error	PIY, SP5
5360	PIX hard error	PIX, SP5

Class 6 subtests—Soft error interrupts

Class 6 subtests verify that each FRU in the system capable of generating a soft error can cause the generation of a soft error interrupt on the service processor, and that the service processor correctly determines the functional unit sending the error. These subtests use scan operations to set and clear a soft error condition on the FRU or FRUs under test.

Note

The CUJ is a possible source of failure for certain subtests.

Subtests 6200-6235—Memory soft error interrupt

Subtests 6200-6235, listed in Table 84, verify the memory FRUs can generate soft error interrupts.

Table 84
Class 6 memory FRU subtests

Subtest	Test performed	Probable fault location
6200	Memory 0 even soft error	ME0, SP5
6205	Memory 0 odd soft error	MO0, SP5
6210	Memory 1 even soft error	ME1, SP5
6215	Memory 1 odd soft error	MO1, SP5
6220	Memory 2 even soft error	ME2, SP5
6225	Memory 2 odd soft error	MO2, SP5
6230	Memory 3 even soft error	ME3, SP5
6235	Memory 3 odd soft error	MO3, SP5

Subtests 6335-6360—Miscellaneous hard error interrupts

Subtests 6335-6360, listed in Table 85, verify miscellaneous FRUs can generate hard error interrupts.

Table 85
Class 6 I/O FRU subtests

Subtest	Test performed	Probable fault location
6335	CUJ soft error	CUJ, SP5
6350	PIY soft error	PIY, SP5
6360	PIX soft error	PIX, SP5

Class 7 subtests—EBUS interface

Class 7 subtests verify the service processor's EBUS interface. These tests check the functionality of the EBUS window map RAM, population map RAM, controller, address translation hardware, and population map. In Table 86, class 7 subtest numbers are grouped into three categories, determined by the second digit of the subtest number.

Subtests in the first group pattern (7000 series) test RAM in the service processor's I/O space.

Subtests in the second group pattern (7100 series) check the functionality of hardware on the service processor.

Subtests in the third group pattern (7200 series) attempt EBUS transactions between the service processor and main memory.

Table 86
Class 7 subtests

Subtest	Test performed	Probable fault location
7000 7010	EBUS window RAM EBUS population map RAM	SP5 SP5
7100 7110	EBUS controller EBUS population map verification	SP5 SP5
7200	EBUS transfer test	SP5 , PIY, ME0, MO0

Subtest 7000—EBUS window RAM

Subtest 7000, listed in Table 86, pattern tests the EBUS window map RAM with alternating zeros and ones (01010101) and alternating ones and zeros (10101010). If these pattern tests pass, an address uniqueness test is executed. The EBUS window map RAM is also tested for parity by writing bad parity to it and reading the result, which should generate a parity error. The EBUS window map RAM is saved before the test starts, and restored after testing completes.

Subtest 7010—EBUS population map RAM

Subtest 7010, listed in Table 86, tests the EBUS population map RAM by completing the following steps:

1. A zero is written to all EBUS population map locations, and the entire EBUS population map is read to ensure that every bit is set.
2. For each location of the EBUS population map, the following occurs:
   a. The location is set to one.
   b. Every EBUS population map location is read and checked to ensure that each bit contains the expected value.
   c. The location is cleared to zero.
3. A one is written to all EBUS population map locations, and the entire EBUS population map is read to ensure that every bit is set.
4. For each location of the EBUS population map, the following occurs:
   a. The location is set to zero.
   b. Every EBUS population map location is read and checked to ensure that each bit contains the expected value.
   c. The location is cleared to one.

The EBUS population map RAM is saved before the test starts and restored after testing completes.

Subtest 7100—EBUS controller

Subtest 7100, listed in Table 86, tests the EBUS controller on the service processor. For each main memory window, two types of checks are made.

First, each EBUS window is configured to map into page zero of ring zero of main memory. The EBUS population RAM is written to indicate all of main memory exists. The functionality of the valid bit is then checked by attempting an invalid access.

Second, for each EBUS window, illegal combinations of the following bits are checked to ensure they are flagged as invalid:

- lw_wr
- scrub
- tac
- msync
- I/O
- tas

It is important to note that this subtest does not test the functionality of these bits, but only the invalidity of certain combinations. The EBUS window and population maps are saved before the test starts, and restored after testing completes.

Subtest 7110—EBUS population map verification

Subtest 7110, listed in Table 86, clears the EBUS population map to indicate that no main memory is available. A main memory read is then attempted to each of the 1024 4-megabyte blocks in main memory. Each read attempt is checked to ensure that it generates a service processor bus error, and does not attempt to actually read RAM memory. The EBUS window and population maps are saved before the test starts, and restored after testing completes.

Subtest 7200—EBUS transfer test

Subtest 7200, listed in Table 86, is designed to ensure that the EBUS windows can read and write to main memory. The test performs the following procedure:

1. Save the EBUS window and population RAM.
2. Set each EBUS window to the first available main memory block, but have each page invalid.
3. Set the population map to indicate only the first main memory block exists.
4. For each EBUS window the following is performed:
   a. Set the valid bit.
   b. Write data to main memory.
   c. Read the data back and verify the result.
   d. Perform a test and set (TAS) operation and verify the result.
   e. Perform a test and clear (TAC) operation and verify the result.
   f. Perform a scrub operation.
   g. Perform a long word write and verify the result.
   h. Reset the valid bit.
5. Restore the EBUS window and population RAM.

Class 8 subtest—Interrupt bus integrity

Class 8 subtest verify system interrupt bus operation. The subtest sets up the service processor as the receiver of interrupts.

Subtest 8000, listed in Table 87, causes the service processor to interrupt itself. Each of the service processor's eight system interrupts is tested in ascending order.

Table 87
Class 8 subtests

Subtest	Test performed	Probable fault location
8000	SP5—SP5 interrupt	SP5 , PI2

Class 9 subtests—Margin tests

The margin subtests verify the SCM or ESM interface, and the ability of the service processor to margin the power supplies and the system clock rate. In Table 88, class 9 subtests are grouped into three categories, determined by the second digit of the subtest number.

Subtests in the first group pattern (9000 series) test communication between the SCM or ESM and the service processor.

Subtests in the second group pattern (9100 series) margin power supplies to ensure the resulting voltage is within test tolerance. For power margining, the third digit of the subtest number indicates the voltage being margined, and the fourth digit indicates the margin level.

Subtests in the third group pattern (9100 series) margin the system clock rate and ensures the resulting frequency is within tolerance. For clock margining, the fourth digit of the subtest number indicates the margin level.

Table 88
Class 9 subtests

Subtest	Test performed	Probable fault location
9010	Check SP5—SM BUS	SP5, SM
9020	Check local and remote	SP5, SM
9100	Check +5 volts at nominal margin	SP5, SM
9104	Check +5 volts at low margin	SP5, SM
9105	Check +5 volts at high margin	SP5, SM
9110	Check +12 volts	SP5, SM
9120	Check -5 volts	SP5, SM
9130	Check -12 volts	SP5, SM
9140	Check -4.5 volts at nominal margin	SP5, SM
9144	Check -4.5 volts at low margin	SP5, SM
9145	Check -4.5 volts at high margin	SP5, SM
9150	Check -2 volts at nominal margin	SP5, SM
9154	Check -2 volts at low margin	SP5, SM
9155	Check -2 volts at high margin	SP5, SM
9160	Check -5.2 volts at nominal margin	SP5, SM
9164	Check -5.2 volts at low margin	SP5, SM
9165	Check -5.2 volts at high margin	SP5, SM
9210	Check normal CPU clocks at nominal margin	SP5, CUJ
9212	Check normal CPU clocks at high margin	SP5, CUJ
9213	Check normal CPU clocks at low margin	SP5, CUJ
9214	Check forced CPU clocks at nominal margin	SP5, CUJ
9216	Check forced CPU clocks at low margin	SP5, CUJ
9217	Check forced CPU clocks at high margin	SP5, CUJ

Subtest 9010—Service processor to SCM/ESM bus

Subtest 9010 checks the service processor to SCM interface bus by writing `0x00` and `0xFF` to the SCM and reading `0x00` and `0xFF` from the SCM. If the SCM detects a data bus failure when written to, it will turn off system power and display an error message on the front panel hex display. Refer to the *CONVEX System Manager's Guide* for more information.

Subtest 9020—Local and remote operation

Subtest 9020 checks the SCM's ability to indicate local and remote operation. By giving commands to the SCM, the local and remote signals are toggled and verified on the service processor.

Subtests 9100-9199—Power supply values

These subtests check each power supply against specified maximum and minimum values for all possible margins.

1. The nominal margin is checked from 97% to 103% of the voltage being tested.
2. The lower margin is checked from 93% to 100% of the voltage being tested.
3. The upper margin is checked from 100% to 107% of the voltage being tested.

The +5, -2, and -4.5 voltage levels are checked at nominal, low, and high margins. The +12, -5, and -12 voltage levels are not marginable and for this reason are only checked for nominal range. The marginable supplies are left in the margin state in which they entered the subtest.

Subtests 9200-9299—Clock frequency

These subtests verify the ability of the service processor to margin the clocks. The clock frequency register is used to select the clock frequency, and the interval timer is utilized to determine the actual frequency. For system clock margining, the system clock is checked to ensure it is within 5 percent of the nominal value.

The clock rate is restored to its initial rate after testing.

Subtest error messages

When any service processor interface subtest fails, a descriptive error message is displayed. The following sections list the error messages, grouped by subtests.

Initialization and general messages

These messages can be issued at any time, but are more likely to be seen at initialization time.

```
spu4000. scn_init call failed - test aborted
```

The file /mnt/usr/scn/scn_rings does not exist, or is corrupt.

```
spu4000. unknown machine class - test aborted
```

The system serial number is invalid, which could result, for example, if this test were executed on the wrong type of system.

```
spu4000. spu4000 has no fault analyzer-test aborted
```

There is an error in the invocation sequence.

```
received unexpected bus error-test aborted
```

```
received segmentation violation - test aborted
```

These test messages can appear at any time the indicated error occurs during testing.

Subtest 1000—Error message

The error message in Figure 73 results from an error in subtest 1000.

Figure 73
Error message for subtest 1000

```
service processor register failed pattern test
register: IIII  pattern: JJJJ
address: 0xKKKKKK  actual: 0xLLLL  expected: 0xMMMM
```

In Figure 73:

`IIII`	Name of the register which failed
`JJJJ`	Pattern which failed
`0xKKKKKK`	Address of the failing register, in hexadecimal
`0xLLLL`	Actual value read, in hexadecimal
`0xMMMM`	Expected value, in hexadecimal

Subtest 1100—Error message

The error message in Figure 74 results from an error in subtest 1100.

Figure 74
Error message for subtest 1100

```
system serial number invalid
actual: 0xIIII  expected: 0x2JJJ, 0x3JJJ, 0x4JJJ, or 0x5JJJ
```

In Figure 74:

`0xIII`	Actual system serial number, in hexadecimal
`0xJJJJ`	Any hexadecimal number

Subtest 1200—Error message

The error messages in Figure 75 result from an error in subtest 1200.

Figure 75
Error messages for subtest 1200

```
busy bit set before hitting go

busy bit set before hitting go

run_arm bit set before hitting go

ecr_not_zero bit clear before hitting go

run_arm bit did not set before timeout

busy bit clear after run_arm bit set

ecr_not_zero bit clear after run_arm bit set

busy bit did not clear before timeout

run_arm bit cleared when busy cleared

ecr_not_zero bit did not clear when busy cleared
```

Subtest 1300—Error message

The error message in Figure 76 results from an error in subtest 1300.

Figure 76
Error message for subtest 1300

```
service processor real time clock not counting
```

Subtest 2000—Error message

The error message in Figure 77 results from an error in subtest 2000.

Figure 77
Error message for subtest 2000

```
scan loopback failed . direction: IIII
shift count: 0xJJJ  actual: 0xKKKK  expected: 0xLLLL odena: 0xM
```

In Figure 77:

`IIII`	"RIGHT" or "LEFT"
`0xJJJ`	Shift count, in hexadecimal
`0xKKKK`	Expected SDR contents, in hexadecimal
`0xLLLL`	Actual SDR contents, in hexadecimal
`0xM`	The odena value used, in hexadecimal

Subtests 2100-2407 or 4100-4407—Error message

The error message in Figure 78 results from an error in a class 2 or class 4 subtest.

Figure 78
Error message for subtests 2100-2407 or 4100-4407

```
total ring length is shorter than bidirectional part
ring_length: 0xIIII (JJJJ) bi_length: 0xKKKK (LLLL)
```

In Figure 78:

`0xIIII`	Total ring length, in hexadecimal
`JJJJ`	Total ring length, in decimal
`0xKKKK`	Bidirectional ring length, in hexadecimal
`LLLL`	Bidirectional ring length, in decimal

The error messages in Figure 79 result from an error in a class 2 or class 4 subtest.

Figure 79
Error messages for subtests 2100-2407 or 4100-4407

```
Unable to get ring into scannable state

scn_rd returned error status II

scn_wr returned error status II
```

In Figure 79:

`II`	Error code

For class 2 subtests only, either of the error messages in Figure 80 result from an error in subtests 2100-2407.

Figure 80
Error message for subtests 2100-2407

```
LSB failure.  wrote I, read J

MSB failure.  wrote I, read J
```

In Figure 80:

I	Binary value written to the ring
J	Binary value read from the ring

For class 4 subtests only, the error messages in Figure 81 and Figure 82 result from errors in subtests 4100-4407.

Figure 81
Error message for subtests 4100-4407

```
scan ring failure in bidirectional part of scan ring isolated failing
bit (0 - 0xII (0 - JJ)): 0xKK (LL)
failure detected when writing and reading in the LEFT direction
isolation done by writing LEFT, reading RIGHT

expected buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right
bit
<buffer, displayed as 16 bit hex values>

actual buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right
bit
<buffer, displayed as 16 bit hex values>
```

Figure 82
Error message for subtests 4100-4407

```
scan ring failure isolated
failing bit (0 - 0xII (0 - JJ)) 0xKK (LL)
failure detected when writing and reading in the RIGHT direction
isolation done by writing RIGHT, reading LEFT

expected buffer.
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right
bit
<buffer, displayed as 16 bit hex values>

actual buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right
bit
<buffer, displayed as 16 bit hex values>
```

In Figure 81 and Figure 82:

`0xII`	MSB, in hexadecimal
`JJ`	MSB, in decimal
`0xKK`	Failing bit, in hexadecimal
`LL`	Failing bit, in decimal
`0xMM`	Number of bits in the buffer, in hexadecimal
`NN`	Number of bits in the buffer, in decimal
`0xWW`	Number of bits displayed in the MSW, in hexadecimal
`XX`	Number of bits displayed in the MSW, in decimal

For class 4 subtests only, either of the error messages in Figure 83 result from an error in subtests 4100-4407.

Figure 83
Error messages for subtests 4100-4407

```
unable to isolate scan ring failure in bidirection part of scan ring
failure detected when writing and reading in the LEFT direction
isolation attempted by writing LEFT, reading RIGHT


unable to isolate scan ring failure
failure detected when writing and reading in the RIGHT direction
isolation attempted by writing RIGHT, reading LEFT
```

Subtests 3100-3407—Error message

The error messages in Figure 84 result from an error in subtests 3100-3407.

Figure 84
Error messages for subtests 3100-3407

```
unable to open data base file /mnt/usr/lib/DB_cop


unable to read cop for slot IIII


type of board installed in slot IIII unknown - part number JJJJ


board type KKKK (part number JJJJ) not allowed in slot IIII
```

In Figure 84:

`IIII`	Name of the slot under test
`JJJJ`	Part number read from the cop
`KKKK`	Board type read from file /mnt/usr/lib/DB_cop

Subtests 3708-3750—Error message

The error messages in Figure 85 result from an error in subtests 3708-3750.

Figure 85
Error messages for subtests 3708-3750

```
miscompare at scan ring address: 0xAAAAAA
actual: 0xBBBBBB  expected: 0xCCCCCC
```

In Figure 86:

`0xAAAAAA`	Scan ring address where the error was found
`0xBBBBBB`	Actual data read from the chip
`0xCCCCCC`	Expected data

Subtests 4600-4617—Error message

The error message in Figure 86 results from an error in subtests 4600-4617.

Figure 86
Error message for subtests 4600-4617

```
miscompare at buffer word AA (bits BB - CC)
masked write   masked read
0xIIII         0xJJJJ
pattern 1 of 4:    all 0's (except for CKE, WE, CS)
pattern 2 of 4:    all 1's (except for CKE, WE, CS)
pattern 3 of 4:    all 0's (except for CKE, WE, CS), with bit 0 a 1
pattern 4 of 4:    all 1's (except for CKE, WE, CS), with bit 0 a 0
```

In Figure 86:

AA	Work number in the scan buffer
BB - CC	Scan ring bit range for this word
0xIIII	Value written to the field
0xJJJJ	Value read from the field

Only one of the last four lines is displayed at a time.

Subtests 4620-4647—Error message

The error messages in Figure 87 result from an error in subtests 4620-4647.

Figure 87
Error message for subtests 4620-4647

```
mod_error returned AA when attempting hard clear of BBBB
```

In Figure 87:

AA	Return value indicating an error
BBBB	Ring that was being processed

The messages in Figure 88 result from a problem with subtests 4620-4647 or a general hardware problem.

Figure 88
Error messages for subtests 4620-4647

```
clock of ring failed

read of ring failed

unable to find field structure for field CCCC

get of field CCC failed

field: CCCC   actual: 0xDDDD  expected: 0xEEEE
```

In Figure 88:

The ring	Ring under test
CCCC	Name of the field
0xDDDD	Value read from the scan ring, in hexadecimal
0xEEEE	Expected value, in hexadecimal

Subtests 5100-5360 or 6103-6360—Error message

The error messages in Figure 89 result if an error occurred while attempting to modify the state of an error in subtests 5100-5360 or 6103-6360.

Figure 89
Error messages for subtests 5100-5360 or 6103-6360

```
mod_error returned I when attempting hard clear of JJJJ

mod_error returned I when attempting soft clear of JJJJ

mod_error returned I on set of JJJJ

mod_error returned I on clear of JJJJ
```

In Figure 89:

`JJJJ`	Name of the ring which failed
`I`	One of the following error codes:
`-1`	Scan error occurred
`1`	Illegal ring for specified function
`4`	Illegal function
`8`	Field not found in scan ring
`16`	Ring has zero length

The error messages in Figure 90 result if an operation to modify the state of an error did not have the desired effect in subtests 5100-5360 or 6103-6360.

Figure 90
Error messages for subtests 5100-5360 or 6103-6360

```
unable to disable all hard errors
actual ESR: 0xIII.  actual SEL: 0xJJJJ


unable to disable all soft errors
actual ESR: 0xIII.  actual SEL: 0xJJJJ


hard error set after clearing error
actual ESR: 0xIII.  actual SEL: 0xJJJJ


soft error set after clearing error
actual ESR: 0xIII.  actual SEL: 0xJJJJ


hard error not set when expected
actual ESR: 0xIII.  actual SEL: 0xJJJJ


soft error not set when expected
actual ESR: 0xIII.  actual SEL: 0xJJJJ


hard error set when only soft should be
actual ESR: 0xIII.  actual SEL: 0xJJJJ


soft error set when only hard should be
actual ESR: 0xIII.  actual SEL: 0xJJJJ


hard error set after (re)clearing error
actual ESR: 0xIII.  actual SEL: 0xJJJJ


soft error set after (re)clearing error
actual ESR: 0xIII.  actual SEL: 0xJJJJ
```

In Figure 90:

`0xIIII`	Actual ESR value, in hexadecimal
`0xJJJJ`	Actual SEL value, in hexadecimal

Subtest 7000—Error message

The error messages in Figure 91 result from an error in subtest 7000.

Figure 91
Error messages for subtest 7000

```
ebus ram data rebound failed - pattern: IIII
```

```
ebus ram parity check failed - pattern: IIII
```

In Figure 91:

`IIII`	Pattern which failed.

Subtest 7010—Error message

The error message in Figure 92 results from an error in subtest 7010.

Figure 92
Error messages for subtest 7010

```
ebus population map ram error detected
address: 0xIIIII.  actual: 0xJJJJ  expected: 0xKKKK
```

In Figure 92:

`0xIIIIII`	Service processor I/O address of the population RAM
`0xJJJJ`	Actual value, in hexadecimal
`0xKKKK`	Expected value, in hexadecimal

Subtest 7100—Error message

The error message in Figure 93 results from an error in subtest 7100.

Figure 93
Error messages for subtest 7100

```
ebus controller functionality error
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ
actual BSR: 0xKKK.  expected BSR: 0xLLLL
```

In Figure 93:

`0xIIIIII`	Service processor I/O address of the window map, in hexadecimal
`0xJJJJJJJJ`	Contents of the window map, in hexadecimal
`0xKKKK`	Actual bus error register, in hexadecimal
`0xLLLL`	Expected bus error register, in hexadecimal

Subtest 7110—Error message

The error message in Figure 94 results from an error in subtest 7110.

Figure 94
Error messages for subtest 7110

```
ebus population map functionality error detected
expected bus error did not occur
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ
```

In Figure 94:

`IIIIII`	Service processor I/O address of the window map, in hexadecimal
`0xJJJJJJJJ`	Contents of the window map, in hexadecimal

Subtest 7200—Error message

The error messages in Figure 95 result from errors in subtest 7200.

Figure 95
Error messages for subtest 7200

```
ebus operation caused unexpected bus error
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ


ebus long word write failed
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ
actual: 0xYYYYYYYYYYYYYYYY  expected: 0xZZZZZZZZZZZZZZZZ


ebus test and clear (TAC) operation failed
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ
data read during TAC: 0xWWWW  data read from memory after TAC: 0xXXXX


ebus test and set (TAS) operation failed
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ data read
during TAS: 0xMMMM  data read from memory after TAS: 0xNNNN


normal (16 bit) ebus read or write failed
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ
data written: 0xKKKK  data read: 0xLLLL


ebus operation caused segmentation violation.
window map address: 0xIIIIII  window map data: 0xJJJJJJJJ
data written: 0xKKKK  data read: 0xLLLL
```

In Figure 95:

0xIIIIII	Service processor I/O address of the window map, in hexadecimal
0xJJJJJJJJ	Contents of the window map, in hexadecimal
0xKKKK	Data written to memory, in hexadecimal
0xLLLL	Data read from memory, in hexadecimal
0xMMMM	Data read during a TAS operation, in hexadecimal

`0xNNNN`	Data read during a TAC operation, in hexadecimal
`0xWWWW`	Actual value of the memory location, in hexadecimal
`XXXX`	Actual value of the memory location, in hexadecimal
`0xYYYYYYYYYYYYYYYY`	Data read from memory, in hexadecimal
`0xZZZZZZZZZZZZZZZZ`	Data written to memory, in hexadecimal

Subtests 8000-8010—Error message

The error messages in Figure 96, Figure 97, and Figure 98 result from errors in subtests 8000-8010.

Figure 96
Error messages for subtests 8000-8010

```
received unexpected interrupt 0xII while testing interrupt 0xJJ
IER: 0xKKKKKKKK
```

In Figure 96:

`0xII`	Interrupt which was received, in hexadecimal
`0xJJ`	Interrupt under test, in hexadecimal
`0xKKKKKKKK`	Contents of the IER, in hexadecimal

Figure 97
Error messages for subtests 8000-8010

```
ISR not as expected while testing interrupt 0xII .
actual: 0xJJJJJJJJ  expected: 0xKKKKKKKK
```

In Figure 97:

`0xII`	Interrupt under test, in hexadecimal
`0xJJJJJJJ`	Actual contents of the ISR, in hexadecimal
`0xKKKKKKKK`	Expected contents of the ISR, in hexadecimal

Figure 98
Error messages for subtests 8000-8010

```
incorrect interrupt received
received: 0xII  expected: 0xJJ
```

In Figure 98 :

`0xII`	Interrupt received, in hexadecimal
`0xJJ`	Interrupt expected, in hexadecimal

The error message in Figure 99 results from an error in subtest 8010 only.

Figure 99
Error messages for subtest 8010

```
PIT never interrupted SP5 while testing interrupt 0xII
```

In Figure 99 :

`0xII` Interrupt under test, in hexadecimal

Class 9 Subtests—Error message

The error messages in Figure 100 and Figure 101 result from errors occurring during a class 9 subtest.

Figure 100
Error messages for class 9 subtests

```
system monitor command failed
address: 0xIIIIII  operation: JJJJ
```

In Figure 100:

`0xIIIIII`	Failing service processor I/O address, in hexadecimal
`JJJJ`	"unknown," "read," or "write"

Figure 101
Error messages for class 9 subtests

```
system monitor command failed during call to IIII
```

In Figure 101:

`IIII`	"meas_voltage ()," "pwr_marg_num ()," or "pwr_marg ()"

Subtest 9010—Error message

The error message in Figure 102 results from an error in subtest 9010.

Figure 102
Error messages for subtest 9010

```
read 0xIIII from system monitor, expected 0xJJJJ
```

In Figure 102:

`0xIIII`	Data read from the SCM or ESM, in hexadecimal
`0xJJJJ`	Expected data, in hexadecimal

Subtest 9020—Error message

The error messages in Figure 103 result from an error in subtest 9020.

Figure 103
Error messages for subtest 9020

```
wouldn't go into local mode, 1st attempt - cpr: 0xIIII

incorrectly in remote mode, 1st attempt - cpr: 0xIIII

wouldn't go into remote mode, 1st attempt - cpr: 0xIIII

incorrectly in local mode, 1st attempt - cpr: 0xIIII

wouldn't go into local mode, 2nd attempt - cpr: 0xIIII

incorrectly in remote mode, 2nd attempt - cpr: 0xIIII

wouldn't go into remote mode, 2nd attempt - cpr: 0xIIII

incorrectly in local mode, 2nd attempt - cpr: 0xIIII

couldn't restore local mode - cpr: 0xIIII

incorrectly in remote mode after restore - cpr: 0xIIII

couldn't restore remote mode - cpr: 0xIIII

incorrectly in local mode after restore - cpr: 0xIIII
```

In Figure 103:

`0xIIII`	Actual control panel value, in hexadecimal

Subtests 9100-9155—Error message

The error message in Figure 104 results from an error in subtests 9100-9155.

Figure 104
Error message for subtests 9100-9155

```
IIII voltage out of tolerance at JJJJ margin
actual: KKK.KK  minimum: LLL.LL  maximum: MMM.MM
```

In Figure 104:

`IIII`	One of: +5 +12 -5 -12 -4.5 -2
`JJJJ`	"nominal", "lower", or "upper"
`KKK.KK`	Actual voltage, in VDC
`LLL.LL`	Minimum allowable voltage, in VDC
`MMM.MM`	Maximum allowable voltage, in VDC

Subtests 9200-9206—Error message

Subtests 9200-9206 verify the ability of the service processor to margin the clocks. The system clock is checked to ensure it is within five percent of a nominal, low, upper, and extended margin.

The error message in Figure 105 results from an error in subtests 9200-9206.

Figure 105
Error message for subtests 9200-9206

```
clock out of tolerance at IIII margin
actual: JJJ.JJ  minimum: KKK.KK  maximum: LLL.LL
```

In Figure 105:

`IIII`	"nominal", "lower", or "upper"
`JJJ.JJ`	Actual frequency in MHz
`KKK.KK`	Minimum allowable frequency, in MHz
`LLL.LL`	Maximum allowable frequency, in MHz

Memory subsystem diagnostic test (mem4100)

9

The memory subsystem diagnostic test (mem4100) verifies the functionality of the memory subsystem. All subtests may be attempted on any legal combination of MCM, MCM2, and MCM3 boards, although not all subtests are applicable to all board types.

In general, simple subtests build to more complex ones. mem4100 begins by scan testing basic and essential circuitry, and progresses to testing EBUS operation and performing pattern tests. Next, it performs scan-based testing of MCM, MCM2, and MCM3 board-specific functionality. It completes with a variety of CPU-based subtests. There are several selectable test parameters.

The memory test is designed to be a tool for use by hardware engineering personnel during development of the memory system, by manufacturing personnel during board testing, and by field personnel during diagnostics testing. This chapter was written specifically to assist the field personnel with diagnostic testing.

This test is executed from the service processor and is initiated under the diagnostic shell (dshell).

Prerequisites and required equipment

This test may be executed on any C3400 Series CPU.

`mem4100` may be executed on a wide variety of hardware configurations, but a minimum configuration is required. This minimum configuration requires the following hardware to be available in the system and operational:

- A service processor (SP5)
- A peripheral interface adapter (PI2)
- CPU utility board(s)
- At least one pair of memory boards, one even and one odd, (MCM, MCM2, or MCM3)

The functional areas tested by `mem4100` are shown in Table 89, which shows an overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run.

Table 89
`mem4100` functional areas tested

Functional area	Tested by diagnostic	Exercised by diagnostic
CPU	No	Yes
CUJ	No	Yes
Memory even	Yes	No
Memory odd	Yes	No
PI2	No	Yes
CCU	No	No
SP5	No	Yes

Caution

If this test is executed and failures occur, the initall utility must be executed prior to any other test invocation. Failure to execute the initall utility in this circumstance could result in invalid test results.

Test invocation

To invoke the mem4100 test, use the procedure shown in Figure 106.

Figure 106
mem4100 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL

: test mem4100 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if mem4100 was executed with failures, or if spu4000 was executed, then initall must be executed prior to test execution. It should be run only after the EPROM based self-test has passed. Failure to execute initall in these circumstances could result in invalid test results.

Entering only

```
test mem4100
```

executes all mem4100 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the -c or -s options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 107.

Figure 107
`mem4100` test parameter menu

```
Test 'mem4100'                                  Thu Nov 19 00:00:00 1986

                    ENTER TEST PARAMETERS

            [ ]     Encloses allowed input ranges or values
            ( )     Encloses the default value
            ^       Returns to the previous prompt
            :nn     Returns to the prompt # nn
            :       Returns to the first unsatisfied prompt
            :?      Reviews previous entries

1. Run default switches [y,n]                                  (y) ->
2. Generate PCM? [y,n]                                         (y) ->
3. Memory pair to test? [0123]                              (0123) ->
4. Run exhaustive subtests? [y,n]                              (n) ->
5. Use CPUs to test memory? [y,n]                              (y) ->
6. CPUs to use? [0123]                                      (0123) ->
7. Load CPU code at beginning or end of memory?[b,e]           (b) ->
8. Halt CPUs on error? [y,n]                                   (n) ->
9. Enter OK, or :NN to return to question NN [OK]           (OK) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

Prompt explanations

A description of the meaning of each prompt follows.

```
1: Run default switches [y,n]                    (y) ->
```

If you respond with **y** or **RETURN**, no additional test parameter prompts are displayed and testing begins.

The following prompts are displayed and answered only if the first prompt is answered with **n**:

```
2: Generate PCM? [y,n]                           (y) ->
```

The default action is for `mem4100` to generate a PCM for the memory subsystem by writing and reading memory in an identical fashion to that of the mminit utility. You may, however, specify not to have a PCM generated by replying **n** to this prompt, causing `mem4100` to use the current service processor PCM.

```
3: Memory pair to test? [0123]                   (0123) ->
```

An available memory pair is both the even and odd memory boards of a given pair. It is not possible to test half of an installed pair, or to test a partially installed pair.

The default action is for `mem4100` to test all available memory pairs. To test a subset of the installed memory pairs, specify the desired pair(s).

```
4: Run exhaustive subtests? [y,n]                (n) ->
```

`mem4100` has several subtests which function mainly as architectural verifiers, but may be useful in finding a bad part or isolating a failure. Currently, only subtests 4200-4207 are exhaustive subtests.

As these subtests take a long time to run, and the hardware tested by them rarely fails, these subtests are not normally executed. To execute these subtests, reply **y** at the prompt.

```
5: Use CPUs to test memory? [y,n]                    (y) ->
```

By default, `mem4100` uses all available CPUs to test memory. It is recommended that you not force the SPU to test memory on systems with large amounts of memory, as the time consumed in service processor-based pattern testing is several orders of magnitude greater than what the CPU subtests require, and coverage is compromised.

To force all memory testing to be done by the service processor, reply **n** to the prompt.

If CPUs are not used to test memory, the next prompts will not appear.

```
6: CPUs to use? [0123]                               (0123) ->
```

By default, `mem4100` uses all available CPUs to test memory. The order of specification is important, as the first CPU used is the first one on the list, and so on; `012` is different from `210` or `021`.

To modify the order or select a subset of the available CPUs, specify the desired CPU(s) in the order desired.

```
7: Load CPU code at beginning or end of memory?[b,e] (b) ->
```

Normally, the code executed by each CPU, as well as the associated data, stack, and PTE pages, are stored in the low memory (the lowest physical addresses). It is possible to locate this information in high memory (the highest physical addresses). This option can be used to allow the CPUs to test low memory, as only the service processor tests the memory the CPUs use to execute. To use high memory for the CPUs to execute from, and allow the CPUs to test low memory, reply **e** at the prompt.

```
8: Halt CPUs on error? [y,n]                         (n) ->
```

Normally, if a CPU detects a data miscompare during a pattern test, it writes a communication structure to a predetermined location in main memory, then polls the communication structure in a loop, waiting for the service processor to tell it to do something different through the same communication structure. To have the CPUs pull halt and immediately stop clocks to the entire system in the event of a data miscompare, reply **y** at the prompt.

```
9. Enter OK, or :NN to return to question NN [OK]    (OK) ->
```

This prompt allows you to change any of the other prompt answers, out of order, if necessary.

Suggestions for using `mem4100`

This section is concerned with which subtests should be run with what parameters, and not with the details of how to run them.

If little is known about the system, or a suspected problem in a system, it is suggested that initially all subtests be executed in sequence, using the default parameter settings.

Should a failure occur in one class of subtests, the cause of the failure should be determined before proceeding to the next class of subtests. Failure to do this may result in the failure of later subtests in unpredictable and misleading ways.

Should all subtests pass and an intermittent problem in the memory system is believed to exist, it is suggested that one of the following be attempted:

1. Execute all class 5 (CPU-based) subtests for each available CPU, with only that one CPU enabled.
2. Loop on all class 5 (CPU-based) subtests, first with all CPUs enabled, then with only one CPU enabled at a time.
3. Loop on subtest 5100, first with all CPUs enabled, then with only one CPU enabled at a time.
4. Enable the exhaustive subtests and rerun the entire diagnostic.

If a problem is suspected in only one pair of memory boards, execute `mem4100` as described above, but modify the user parameter to test only that pair of memory boards.

If a problem is suspected in low memory only, execute `mem4100` as described above, but modify the user parameter to place the CPU code at the high end of memory.

If a problem is suspected with the error correcting code circuitry, run classes 3 and 4, with exhaustive subtests enabled.

Class descriptions

There are five classes of subtests in `mem4100`:

1. Arbitration and crossbar subtests
2. SPU-based basic functionality subtests
3. SPU-based exception functionality subtests (MCM and MCM2)
4. SPU-based exception functionality subtests (MCM3)
5. CPU-based main memory subtests

Subtest descriptions

The following are the various types of subtests, in the approximate order in which subtests execute.

1. Arbitration and crossbar functionality
2. Basic operation functionality
3. ECC functionality
4. Data parity functionality
5. Quick pattern test
6. Intermediate pattern test
7. Exhaustive pattern test

Class 1 subtests—Arbitration and crossbar

Class 1 subtests test the arbitration and crossbar circuitry required for any other subtests to run.

These subtests assume only that the initial test requirements have been met. All other classes of subtests assume the class 1 subtests have passed. If these subtests do not pass, the problem must be fixed before proceeding.

All class 1 subtests apply to MCM, MCM2, and MCM3 memory boards.

Subtest 1025—Arbitration win logic

Subtest 1025 tests the arbitration gate array and arbitration circuitry on each memory board to be tested.

This subtest verifies that the arbitration gate array correctly chooses between the 5 ports (A through E) during write operations. Included are the following combinations:

- All ports individually
- Port E with each of the others, A through D
- Ports A and B
- Ports C and D
- Ports A through D
- Ports A through E

This subtest is executed on each memory card with ECC and parity checking disabled.

Subtest 1200—Crossbar read and write latching tests

Subtest 1200 tests the arbitration gate array and arbitration circuitry on each memory board to be tested.

This subtest verifies the arbitration gate array correctly selects the win sequence between the five ports on reads. It also verifies the read and write pointers in each of the crossbar gate arrays.

This subtest is executed on each memory card with ECC and parity checking disabled.

Class 2 subtests—SPU-based basic functionality

Class 2 subtests start out testing the basic operations the memory system is capable of, and progress to SPU-based memory pattern tests.

All Class 2 subtests apply to MCM, MCM2, and MCM3 memory boards.

Subtests 2000, 2020, and 2030 are the SPU-based equivalents of the CPU-based subtests 5000, 5020, and 5030, respectively.

Subtest 2100 is the SPU-based equivalent of the CPU-based subtests 5100, 5110, and 5120. This subtest performs an address equal data pattern over all of available memory, on a 32-bit word basis.

Subtests 2200, 2210, 2220, and 2230 are the SPU-based equivalents of the CPU-based subtests 5200, 5210, 5220, and 5230.

Subtests 2300, 2310, 2320, and 2330 are the SPU-based equivalents of the CPU-based subtests 5300, 5310, 5320, and 5330.

Subtest 2000—SPU zone bit functionality (16- and 64-bit writes)

Subtest 2000 verifies the ability of the service processor to read and write main memory over the EBUS with various zone and cycle combinations.

This subtest verifies that 16- and 64-bit writes work correctly to each bank of each 16 megabyte block of memory. The subtest first does a longword write to memory and verifies it wrote correctly, then verifies that each 16-bit halfword of the longword can be modified independently of other memory.

Subtest 2010—SPU SCRUB operations

Subtest 2010 verifies the ability of the service processor to perform a SCRUB operation to main memory over the EBUS.

This subtest verifies that SCRUB operations work correctly in each bank of each 16 megabyte block of memory. The subtest first does a longword write to initialize memory with a known value, then does a SCRUB operation with a different value and verifies the memory has not changed.

Subtest 2020—SPU test-and-set operations

Subtest 2020 verifies the ability of the service processor to perform test-and-set operations in main memory over the EBUS.

This subtest verifies that 16-bit test-and-set operations work correctly in each bank of each 16 megabyte block of memory. The subtest first does a longword write to initialize memory with a known value, then does a test-and-set operation, verifies the data returned was correct, and the current memory contents are as expected.

Subtest 2030—SPU test-and-clear operations

Subtest 2030 verifies the ability of the service processor to perform test-and-clear operations in main memory over the EBUS.

This subtest verifies that 16-bit test-and-clear operations work correctly in each bank of each 16 megabyte block of memory. The subtest first does a longword write to initialize memory with a known value, then does a test-and-clear operation, verifies the data returned was correct, and the current memory contents are as expected.

Subtest 2100—SPU address = data pattern

Subtest 2100 is designed to test for address and data bit malfunctions, as well as full functionality of all parts of the memory subsystem.

This subtest performs an address-equals-data pattern over the memory required for the CPU subtests to execute (or all of memory, if CPUs will not be used to test memory). The subtest writes the address of each 32-bit word to each 32-bit word (using 16-bit writes), then reads each 32-bit location (using 16-bit reads) and verifies it contains the correct data. Testing stops when the maximum allowable number of errors is exceeded.

Before performing the pattern test, a level 0 memory system default and a level 0 peripheral interface adapter default are performed. If an error occurs in either of these subtests, an error message is displayed and the subtest ends. Also, if an error occurs while looking up the patterns for this test, turning on the clocks to the memory and I/O subsystems, or initializing parity for the SPU-based pattern subtests, an error message is displayed and the subtest ends.

Subtests 2200-2230—SPU MATS+ memory pattern

Subtests 2200, 2210, 2220, and 2230 are designed to test for address and data bit malfunctions, as well as full functionality of all parts of the memory subsystem.

These subtests perform a MATS+ memory pattern test over the memory required for the CPU subtests to execute (or all of memory, if CPUs will not be used to test memory). Each subtest performs the identical sequence on the same range of memory, with only the data patterns being different.

Subtests 2300-2330—SPU Nair, Thatte, and Abraham's memory pattern

Subtests 2300, 2310, 2320, and 2330 are designed to test for address and data bit malfunctions, as well as full functionality of all parts of the memory subsystem.

These subtests perform a Nair, Thatte, and Abraham's memory pattern test over the memory required for the CPU subtests to execute (or all of memory if CPUs will not be used to test memory). Each subtest performs the identical sequence on the same range of memory, with only the data patterns being different.

Class 3 subtests—SPU-based exception functionality

Class 3 subtests test the error detection and correction, and the parity generation and checking circuitry on each MCM or MCM2 to be tested. Similar MCM3 functionality is tested by the class 4 subtests.

The class 3 subtests assume the class 1 subtests have passed. It is also advisable to run the class 2 subtests first, especially the basic operation subtests (subtests 2000-2030). Occasionally, however, the class 3 subtests can help find the cause of the class 2 pattern test failure.

Subtest 3020—Arbitration win queue

Subtest 3020 tests the arbitration gate array and arbitration circuitry on each MCM or MCM2 memory board to be tested.

This subtest pattern tests the arbitration gate array win queue. The subtest first tests constant patterns to set and clear all bits, and then performs an address uniqueness test.

This subtest is executed on each MCM or MCM2 memory board (up to eight possible) with error correcting code (ECC) and parity checking disabled.

Subtest 3150—Normal ECC and parity circuitry

Subtest 3150 verifies the normal passage of data through the ECC and parity circuitry of MCM or MCM2 memory boards. It enables ECC and writes a data pattern of walking ones via scan operations. It then performs scan read operations and verifies that no errors are detected, and that the data read equals the data written.

This subtest tests the eight ECC and eight parity generators and checkers on each MCM or MCM2 memory boards to be tested.

This subtest is executed with ECC and parity enabled.

Subtest 3151—Write parity error detection

Subtest 3051 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the ability of MCM or MCM2 memory boards to detect parity errors on write data. A data pattern of walking ones for each byte and parity of all zeros is written. This causes a parity error for each of the parity bits of the 4 bytes. The subtest verifies that the tested MCM or MCM2 memory boards detect the error by checking that the hard error interrupt occurs and that the proper byte is tagged as being in error.

This subtest tests the eight parity generators and checkers on each MCM or MCM2 memory board cards to be tested. This subtest is executed with ECC and parity enabled.

Subtest 3152—Single bit ECC, data bits

Subtest 3052 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the ability of MCM or MCM2 memory boards to detect and correct single-bit ECC errors. It writes a data pattern of walking ones that has the ECC codes of a 0 data pattern. The pattern is written via scan operations. The ECC code is forced on writes which places the bad ECC code in memory. The subtest then performs a scan read operation. A verification is made that the correct single-bit error is detected for all 32 bits through the soft error interrupt of the service processor and the soft error scan ring information of the memory soft log. The test also verifies that the data read equals all zeros, since the bit is toggled. The data is then reread and it is reverified that the soft error occurs, since the data in memory is not corrected.

The test is then executed with the ECC code forced on reads. The same results are expected. This verifies the ECC generation logic on writing the data and the generation of the ECC code used to check what was written in memory against what was read.

The next data pattern is a walking 0 with an ECC code for all ones. The same results are expected.

This subtest tests the eight ECC generators and checkers for each MCM or MCM2 memory boards to be tested. This subtest is executed with ECC and parity enabled.

Subtest 3153—Single bit ECC, check bits

Subtest 3153 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the ability of MCM or MCM2 memory boards to detect and correct single-bit check bit ECC errors. The subtest uses a pattern of all zeros data and the forced ECC code is selected to cause an ECC error in each of the check-bits. The subtest verifies that the correct single-bit error is detected for all seven check-bits through the soft error interrupt of the service processor and the soft error scan ring information of the memory soft log. It also verifies the data read is all zeros (since the error is in the check bits). The data is reread and the same results are expected.

The test is then executed with the ECC code forced on reads. The same results are expected. This verifies the ECC generation logic on writing the data and the generation of the ECC code used to check what was written in memory against what was read.

The next data pattern is ones with ECC codes selected to cause an ECC check-bit error. The test verifies that the bit in error can be toggled in either direction.

This subtest tests the eight ECC generators and checkers on each MCM or MCM2 to be tested. This subtest is executed with ECC and parity enabled.

Subtest 3154—Double bit ECC, data bits

Subtest 3154 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the ability of MCM or MCM2 memory boards to detect double-bit data bit ECC errors. The subtest writes a pattern similar to walking ones, except it has two bits set. An ECC code for 0 data is forced on writes that places the bad ECC code in memory. The subtest then performs a main memory read operation via scan. A verification is made that the correct double-bit error is detected for all 32-bits through the hard error interrupt of the service processor and the hard error scan ring information of the memory soft log. Hard error interrupts are expected, and multibit error information is expected in the scan results. In this subtest, data is not checked.

This subtest tests the eight ECC generators and checkers on each MCM or MCM2 to be tested. This subtest is executed with ECC and parity enabled.

Subtest 3155—Double bit ECC, check bits

Subtest 3155 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the ability of the memory system to detect double-bit check bit errors. It uses a forced ECC code, which causes a double-bit error. Hard error interrupts are expected and multi-bit error information is expected in the scan results. In this subtest, data is not checked.

This subtest tests the eight ECC generators and checkers for the MCM or MCM2 memory boards to be tested.

Subtest 3156—Single bit ECC, partial writes

Subtest 3156 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest checks for correct operation of MCM or MCM2 memory board single-bit ECC when partial writes are performed. The subtest uses a pattern of constant 1 with an ECC code for all zeros.

The three test conditions are as follows:

- Check results when pattern forces bad ECC on writes twice, then read and check data.
- Check results when test forces bad ECC code on first write, uses normal ECC on second write, and then read data.
- Check results when test forces bad ECC code on first write, use normal ECC on next two writes and then read data.

For the first case, the subtest expects normal return from the first write, since it gets the result of the first read returned. It then expects the correct soft error information on the second write which gets the result of the first write as read data. The subtest also expects soft errors on the read operation as it is getting the result of the second write returned when it reads. The data is then checked for all zeros.

For the second case, the subtest expects normal return from the first write, since it gets the result of the first read returned. It then expects the correct soft error information on the second write which gets the result of the first write as read data. Since the second write is performed with good ECC, the subtest expects normal return with no ECC errors on the read operation. The data is then checked for the constant 1 pattern.

For the third case, the subtest expects normal return from the first write, since it gets the result of the first read returned. It then expects the correct soft error information on the second write which gets the result of the first write as read data. Since the second write is performed with good ECC, the subtest expects normal return with no ECC errors on the third write operation. Since the third write is performed with good ECC, the subtest expects normal return with no ECC errors for the read operation. The data is then checked for the constant 1 data.

This subtest is executed with ECC and parity enabled.

Subtest 3157—Single bit ECC (TAM)

Subtest 3157 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest checks for correct ECC operation when test and modify operations are performed on MCM or MCM2. The subtest uses a pattern of constant 1, with an ECC code for all zeros.

The three test conditions are:

- Check results when pattern forces bad ECC on test and modifies twice, then read and check data.
- Check results when test forces bad ECC code on first test and modify, use normal ECC on second test and modify, and then read data.
- Check results when test forces bad ECC code on first test and modify, uses normal ECC on next two test and modifies, and then reads data.

For the first case, the subtest expects normal return from the first test and modify since it gets the result of the first read returned. It then expects the correct soft error information on the second test and modify, which gets the result of the first test and modify as read data. The subtest also expects soft errors on the read operation, as it is getting the result of the second test and modify returned when it reads. The data is then checked for all zeros.

For the second case, the subtest expects normal return from the first test and modify, since it gets the result of the first read returned. It then expects the correct soft error information on the second test and modify, which gets the result of the first test and modify as read data. Since the second test and modify is performed with good ECC, the subtest expects normal return with no ECC errors on the read operation. The data is then checked for the constant 1 pattern.

For the third case, the subtest expects normal return from the first test and modify since it gets the result of the first read returned. It then expects the correct soft error information on the second test and modify, which gets the result of the first test and modify as read data. Since the second test and modify is performed with good ECC, the subtest expects normal return with no ECC errors on the third test and modify operation. Since the third test and modify is performed with good ECC, the subtest expects normal return with no ECC errors for the read operation. The data is then checked for the constant 1 data.

This subtest is executed with ECC and parity enabled.

Subtest 3158—SCRUB operation

Subtest 3158 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the SCRUB operation on MCM or MCM2 memory boards. First, a pattern of walking ones is written to memory with forced bad ECC codes via scan. The subtest then reads from memory via scan and verifies the correct soft error occurs and the correct data. The subtest then performs a SCRUB operation to memory, rereads the data and verifies that a normal return occurs, after which the data is verified.

This subtest is executed with ECC and parity enabled.

Subtest 3160—Normal ECC and parity circuitry

Subtest 3160 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the normal passage of data through the ECC and parity circuitry of MCM or MCM2 memory boards. It enables ECC and writes a pattern of walking ones data via EBUS transfers. It then performs read operations via EBUS transfers and verifies that no errors are detected and that the data read equals the data written.

This subtest tests the eight ECC and parity generators and checkers on each MCM or MCM2 to be tested. This subtest is executed with ECC and parity enabled.

Subtest 3161—Read parity error detection

Subtest 3161 tests the ECC and parity circuitry on each MCM or MCM2 to be tested.

This subtest verifies the ability of the service processor to detect parity errors on reads from MCM or MCM2 memory boards. The subtest writes data with good parity to memory via the EBUS. It then sets the forced read parity-bit of the scan ring and performs a read operation via the EBUS. The subtest verifies that the service processor EBUS logic detects the parity error.

This subtest is executed with ECC and parity enabled.

Class 4 subtests—SPU-based exception functionality (MCM3)

Class 4 subtests test the error detection and correction, and the parity generation and checking circuitry on each MCM3 to be tested. Similar MCM or MCM2 functionality is tested by the class 3 subtests.

The class 4 subtests assume the class 1 subtests have passed. It is also advisable to run the class 2 subtests first, especially the basic operation subtests (subtests 2000-2030). Occasionally, however, the class 4 subtests can help find the cause of the class 2 pattern test failure.

Subtests 4000-4007—ECC functionality

Subtests 4000-4007 are the standard subtests, and each tests one single and one multiple bit ECC error in each BECC. These subtests each test one hard and one soft error in each BECC on the appropriate MCM3. Each of these subtests tests one MCM3 memory board, with the board being matched by the last digit of the subtest number.

Subtests 4200-4207—ECC functionality

Subtests 4200-4207 test every possible error on each BECC.

These subtests are exhaustive subtests, and only run if the option to run exhaustive subtests is specified during parameter entry. Each of these subtests verifies that each BECC can detect every detectable single and multiple bit ECC error. These subtests take a long time to execute, and should only be run by manufacturing, or if an ECC failure is suspected.

Each of these subtests tests one MCM3 memory board, with the board being matched by the last digit of the subtest number.

Subtests 4100-4107—Read and write parity

Each subtest verifies the functionality of the parity generators and checkers for read and write data on a specific MCM3 memory board.

Each subtest tests one MCM3, verifying that each bank with memory installed can detect write parity errors. Each subtest also verifies the read parity generation circuitry by having each bank send bad parity to the service processor on a read. Each of the four parity bits on an MCM3 is checked separately, for both reads and writes. Each of these subtests tests one MCM3 memory board, with the board being matched by the last digit of the subtest number.

Class 5 subtests—CPU-based main memory

Class 5 subtests use one or more CPUs to test the memory subsystem. These subtests all assume that the CPUs used for testing are fully functional, and that any failure is the result of a problem in the memory subsystem. These subtests also assume that all class 1 through class 4 subtests have passed. The results of these class 5 subtests may be invalid if either of the above conditions is not true.

CPU testing tests most, but not all of memory at any one time, as the memory required to execute the CPU test is not tested by the CPU. The location of this untested memory is somewhat user selectable. You may specify that it be at the beginning or end of physical memory. Interleaving will affect the actual distribution, and will spread the space across board pairs, if interleaving is greater than 8-way.

All class 5 subtests apply to MCM, MCM2, and MCM3.

The CPUs to use when testing memory are obtained from the order dependent list specified at the beginning of testing.

Subtests 5000, 5020, and 5030 are the CPU-based equivalents of the SPU-based subtests 2000, 2020, and 2030, respectively.

Subtests 5100, 5110, and 5120 are the CPU-based equivalents of the SPU-based subtest 2100.

Subtests 5200, 5210, 5220, and 5230 are the CPU-based equivalents of the SPU-based subtests 2200, 2210, 2220, and 2230.

Subtests 5300, 5310, 5320, and 5330 are the CPU-based equivalents of the SPU-based subtests 2300, 2310, 2320, and 2330.

Subtest 5000—CPU zone bit functionality (8-, 16-, 32-, and 64-bit writes)

Subtest 5000 tests the ability of a CPU to execute all possible memory operations of all legal sizes.

This subtest verifies that all sizes (8, 16, 32, and 64-bit) of writes work on all address boundaries. Each write is attempted 8 times, with the byte address being incremented by one between each attempt. Each 16 megabyte block of memory not required for the CPU test itself is tested, with each block being fully tested before the next block tests starts. Only the first available CPU is used in this subtest.

Subtest 5020—CPU test-and-set operations

Subtest 5020 tests the ability of a CPU to execute all possible memory operations of all legal sizes.

This subtest verifies that a byte test-and-set from the CPU works correctly. A test-and-set is attempted on each byte of a longword, for each 16 megabyte block of memory not required for the CPU test itself. The data is initialized before, and checked after, each test-and-set. Only the first available CPU is used in this subtest.

Subtest 5030—CPU test-and-clear operations

Subtest 5030 tests the ability of a CPU to execute all possible memory operations of all legal sizes.

This subtest verifies that a byte test-and-clear from the CPU works correctly. A test-and-clear is attempted on each byte of a longword, for each 16 megabyte block of memory not required for the CPU test itself. The data is initialized before, and checked after, each test-and-clear. Only the first available CPU is used in this subtest.

Subtests 5100-5120—Address = data memory pattern

Subtests 5100, 5110, and 5120 perform an address equal data pattern over all of available memory, on a 32-bit word basis. They are designed to test for address and data bit malfunctions, as well as full functionality of all parts of the memory subsystem.

Subtest 5100 uses the first available CPU to write all locations, then the same CPU to verify each location has the expected data.

Subtest 5110 splits the memory to be tested into two halves, then has each CPU write one of the halves. After the writes have completed, each CPU verifies the half the other CPU wrote (the half it did not write).

Subtest 5120 is similar to subtest 5110, except that three or more CPUs are involved. Memory is divided into blocks of equal size, based on the number of CPUs available, and each CPU writes one block. When all writes have completed, each CPU verifies the block the previous CPU wrote.

Subtests 5200-5230—CPU MATS+ memory pattern

Subtests 5200, 5210, 5220, and 5230 are designed to test for address and data bit malfunctions, as well as full functionality of all parts of the memory subsystem.

These subtests perform a MATS+ memory pattern test over all of available memory, on a 64-bit longword basis. Each of the subtests uses all available CPUs to perform the test, dividing the memory to be tested into blocks of equal size, based on the number of processors available. Each CPU tests a block of memory, using the MATS+ algorithm. Each subtest performs the identical sequence on the same range of memory, with only the data patterns being different.

Subtests 5300-5330—CPU Nair, Thatte, and Abraham's memory pattern

Subtests 5300, 5310, 5320, and 5330 are designed to test for address and data bit malfunctions, as well as full functionality of all parts of the memory subsystem.

These subtests perform a Nair, Thatte, and Abraham's memory pattern test over all of available memory, on a 64-bit longword basis. Each of the subtests uses all available CPUs to perform the test, dividing the memory to be tested into blocks of equal size, based on the number of processors available. Each CPU tests a block of memory, using the Nair, Thatte, and Abraham algorithm. Each subtest performs the identical sequence on the same range of memory, with only the data patterns being different.

Subtest error message header

Figure 108 shows the header format used to precede `mem4100` error messages.

Figure 108
Error message header for mem4100

```
****. Thu Dec 30 09:24:0. *****
Test.  mem4100.t 1.6  Class: 2  Subtest: 3152 1.3  Count: 1  Error: 0
Failed. Scan testing of single bit ECC detection. Data Bits
```

The first two lines include:

- Day, date, time and year
- Test name (for example, `mem4100.t`)
- Test revision number (for example, `1.6`)
- Subtest class (for example, `2`)
- Subtest number (for example, `3152`)
- Subtest revision number (for example, `1.3`)

The count and error information can be ignored.

The third line contains the `Failed.` header, followed by the subtest name or description.

Example:

```
Scan testing of single bit ECC detection. Data Bits
```

The test error message follows this header. See the following sections for details of the messages, according to the subtest being executed.

Class 1 and 3 subtests—Error messages

The first three lines of all `mem4100` error messages are described previously in the "Subtest error message header" section.

Example class 1 and class 3 error messages are shown in Figure 109.

Figure 109
Example error message for class 1 and class 3

```
Checking forced write ECC codes
Pattern: Walk 1; ECC for data = 0
Slot: 4 Bank: 0
loop cnt: 0
 FAIL   OFFSET/
 TYPE   ADDR      EXP         ACT       COMMENTS
ECC   08000000  08000000   40000000  MCM ecc error addr

paused at fail in subtest 152
```

The first four lines in Figure 109 are the set-up information, and contain what the test was testing when it failed. In this case:

- The test was checking forced write ECC codes.
- The pattern was `Walk 1`.
- The ECC for data was `0`.
- The slot was `4`.
- The memory bank was `0`.
- The loop count was `0`.

The next lines contain the actual error message information. The following headers are used:

- `FAIL TYPE`
- `OFFSET/ADDR` (offset address)
- `EXP` (expected value)
- `ACT` (actual value)
- `COMMENTS` (describe the failure)

The last two lines give a brief explanation of the failure. The FAIL TYPE in the actual error information line may be any of the following:

Fail type	Description
DATA	Data value miscompared
PARITY	Parity value miscompared
ECC	Error correcting code (ECC) miscompared
MM WR	Unexpected return code from an EBUS write operation
MM RD	Unexpected return code from an EBUS read operation
IER	Unexpected interrupt error register
ESR	Unexpected interrupt source register
MM TAM	Unexpected return code from an EBUS test and modify operation
ADDR	Address returned in error

The COMMENTS portion of the error message information gives further explanation of what the test expected or did not expect when the test failed. There are 18 possible comment lines, each of which are listed and explained here.

Comments	Description
hard_err failure	Hard error scan field returned wrong data
par_err failure	Parity error scan field returned wrong data
wr_dat failure	Write data scan field returned wrong data
wr_par failure	Write parity scan field returned wrong data
log cycle type failure	Cycle type scan field returned wrong data

Comments	Description
xfr from MM	Unexpected status from an EBUS transfer on an EBUS write operation
xfr to MM	Unexpected status from an EBUS transfer on an EBUS read operation
TAM of MM	Unexpected status from an EBUS transfer on a test and modify of memory operation
MCM hard err type	MCM had unexpected hard error type
MCM soft err type	MCM had unexpected soft error type
SEL: no memory soft error	Failure to receive memory soft error in soft error log register
MCM ECC value	Unexpected value for ECC code
ISR:no soft err	Failure to receive soft error in interrupt status register
ISR: no soft err expected	Soft error received by interrupt status register when no error was expected
ISR: no hard error	Failure to receive hard error in ISR register
ESR: no MCM error	Failure to receive MCM error in ESR register
ISR: unable to clear soft err	Unable to clear ISR soft error
ISR: unable to clear hard err	Unable to clear ISR hard error

Class 2 subtests—Error messages

The first three lines of all `mem4100` error messages are described previously in the "Subtest error message header" section.

Subtest 2000—Error messages

An example subtest 2000 error message is shown in Figure 110.

Figure 110
Example error message for subtest 2000

```
operation failed:   LW address: 0xAAAAAAAA
previous:   0xBBBB 0xBBBB 0xBBBB 0xBBBB
write data: 0xCCCC to <DD..DD> of the longword (address + E)
current:          0xFFFF 0xFFFF 0xFFFF 0xFFFF
```

In Figure 110 the following fields have specific meaning:

Field	Description
`0xAAAAAAAA`	The physical main memory address, 8-way interleaved.
`0xBBBB`	The contents of memory, before this operation occurred.
`0xCCCC`	The data written to memory.
`DD..DD`	The bits that should have been affected by the write.
`E`	The address offset for the write, in bytes.
`0xFFFF`	The incorrect actual data read from memory.

Subtest 2010—Error messages

An example subtest 2010 error message is shown in Figure 111.

Figure 111
Example error message for subtest 2010

```
operation failed:   LW address: 0xAAAAAAAA
previous:    0xBBBB 0xBBBB 0xBBBB 0xBBBB
write data: SCRUB on <15..0> of the longword
current:           0xFFFF 0xFFFF 0xFFFF 0xFFFF
```

In Figure 111 the following fields have specific meaning:

Field	Description
0xAAAAAAAA	The physical main memory address, 8-way interleaved.
0xBBBB	The contents of memory, before this operation occurred.
0xFFFF	The incorrect actual data read from memory.

Subtest 2020—Error messages

Example subtest 2020 error messages are shown in Figure 112.

Figure 112
Example error messages for subtest 2020

```
test-and-set operation returned unexpected data:LW address: 0xAAAAAAAA
previous:    0x5555
write data: 0xCCCC       (should be previous)
current:     0xFFFF      (should be 0xffff)
```

```
operation failed: LW address: 0xAAAAAAAA
previous:    0xBBBB 0xBBBB 0xBBBB 0xBBBB
write data: TAS on <15..0> of the longword
current:           0xFFFF 0xFFFF 0xFFFF 0xFFFF
```

In Figure 112 the following fields have specific meaning:

Field	Description
0xAAAAAAAA	The physical main memory address, 8-way interleaved.
0xBBBB	The contents of memory, before this operation occurred.
0xCCCC	The data written to memory.
0xFFFF	The incorrect actual data read from memory.
0xFFFF	The correct data.

Subtest 2030—Error messages

Example subtest 2030 error messages are shown in Figure 113.

Figure 113
Example error messages for subtest 2030

```
test-and-set operation returned unexpected data:LW address: 0xAAAAAAAA
previous:   0x5555
write data: 0xCCCC      (should be previous)
current:    0xFFFF      (should be 0xffff)
```

```
operation failed:   LW address: 0xAAAAAAAA
previous:   0xBBBB 0xBBBB 0xBBBB 0xBBBB
write data: TAS on <15..0> of the longword
current:          0xFFFF 0xFFFF 0xFFFF 0xFFFF
```

In Figure 113 the following fields have specific meaning:

Field	Description
0xAAAAAAAA	The physical main memory address, 8-way interleaved.
0xBBBB	The contents of memory, before this operation occurred.
0xCCCC	The data written to memory.
0xFFFF	The incorrect actual data read from memory.
0xFFFF	The correct data.

Subtest 2100-2330—Error messages

Example subtest 2100-2330 error messages are shown in Figure 114.

Figure 114
Example error messages for subtests 2100-2330

```
mcm_default (0xEE, 0) failed
```

```
pia_default (0) failed
```

```
unknown subtest number SSSS
```

```
miscompare at physical address: 0xAAAAAAAA
actual high 16 bits:    0xBBBB      actual low 16 bits:    0xBBBB
expected high 16 bits:  0xCCCC      expected low 16 bits:  0xCCCC
[ location is: DDDDDDDD ]
```

If the failure is a failed memory chip, following a data miscompare error there is a list of possible memory devices. Example subtest 2100-2330 error messages are shown in Figure 115.

Figure 115
Example error messages for subtests 2100-2330

```
received unexpected hard error - ESR is 0xHH
```

```
received unexpected soft error - SEL is 0xHH
```

If a hard or soft error occurs, the hard_logger is automatically invoked after the message is displayed, and the test exits after the hard_logger completes.

In Figure 114 and Figure 115 the following fields have specific meaning:

Field	Description
0xAAAAAAAA	The physical main memory address, 8-way interleaved.
0xBBBB	The current contents of memory.
0xCCCC	The expected contents of memory.
DDDDDDDD	The place in the algorithm where the error was detected.
0xEE	The mask specifying which memory boards to initialize.
SSSS	The subtest number.
GGGGGGGG	The name of the subtest
0xHH	The current value of the indicated register.

Class 4 subtests—Error messages

The first three lines of all `mem4100` error messages are described previously in the "Subtest error message header" section.

The lines that follow are typical of class 4 error messages.

Subtests 4000-4007 and 4200-4207—Error messages

Example subtest 4000-4007 and 4200-4207 error messages are shown in Figure 116.

Figure 116
Example error messages for subtests 4000-4007 and 4200-4207

```
AAAAAAAA soft error
AAAAAAAA hard error
error detected BBBBBBBB
failing address: 0xCCCCCCCC (spu: 0xDDDDDDDD  map: 0xEEEEEEEE
block addresses 0xHHHHHHHH and 0xIIIIIIII  loop:JJ  bank: KK
bad ECC bits:      0xLL  bad data bits: 0xMMMMMMMM
set ECC bits:      0xNN  set data bits: 0xOOOOOOOO
```

Subtests 4100-4107—Error messages

Example subtest 4100-4107 error messages are shown in Figure 116.

Figure 117
Example error messages for subtests 4100-4107

```
AAAAAAAA soft error
AAAAAAAA hard error
error detected BBBBBBBB
failing address: 0xCCCCCCCC (spu: 0xDDDDDDDD  map: 0xEEEEEEEE
block addresses 0xHHHHHHHH and 0xIIIIIIII  loop:JJ  bank: KK
good parity would have been: 0xL  parity used: 0xM
```

Class 5 subtests—Error messages

The first three lines of all `mem4100` error messages are described earlier in the "Subtest error message header" section.

Error message building blocks

Memory mapping

In many error messages the main memory mapping used by the CPUs is displayed. Example class 5 error messages are shown in Figure 118.

Figure 118
Example error messages for class 5 memory mapping

```
address mapping data:
      PS/PE - physical start/end, VS/VE - eirtual start/end
      input information:
            prog_pages: AA  where: BB  array_size: CC
      output information:
            num_to_test: DD  num_blks: EE
            PS: 0xHHHHHHHH  PE: 0xIIIIIIII
      block mapping:
            0:    PS: 0xJJJJJJJJ    VS: 0xKKKKKKKK  VE: 0xLLLLLLLL
            1:    PS: 0xJJJJJJJJ    VS: 0xKKKKKKKK  VE: 0xLLLLLLLL
            2:    PS: 0xJJJJJJJJ    VS: 0xKKKKKKKK  VE: 0xLLLLLLLL
            . . .
```

In Figure 118 the following fields have specific meaning:

Field	Description
AA	The number of pages required for CPU code and data.
BB	Where the code is located (beginning or end or memory)
CC	The maximum number of blocks that can be mapped.
DD	The number of PCM blocks mapped for testing.

Field	Description
EE	The number of entries in the block mapping table.
0xHHHHHHHH	The physical address of the first page of untestable memory.
0xIIIIIIII	The physical address of the last page of untestable memory.
0xJJJJJJJJ	The physical address of the first page of a testable block.
0xKKKKKKKK	The virtual address of the first page of a testable block.
0xLLLLLLLL	The virtual address of the last page of a testable block.

CPU register dump

In many nonmiscompare error messages a register dump is done for an individual CPU, or for each CPU used by the subtest. An example class 5 register dump is shown in Figure 119.

Figure 119
Example error messages for class 5 CPU register dump

```
Register dump for cpu: N
a0: MMMMMMMM  s0: MMMMMMMM MMMMMMMM  t0: MMMMMMMM   pc: MMMMMMMM
a1: MMMMMMMM  s1: MMMMMMMM MMMMMMMM  t1: MMMMMMMM  psw: MMMMMMMM
a2: MMMMMMMM  s2: MMMMMMMM MMMMMMMM  t2: MMMMMMMM  ipc:      MMM
a3: MMMMMMMM  s3: MMMMMMMM MMMMMMMM  t3: MMMMMMMM  ccr:   MMMMMM
a4: MMMMMMMM  s4: MMMMMMMM MMMMMMMM  t4: MMMMMMMM  cir: MM  tid: MM
a5: MMMMMMMM  s5: MMMMMMMM MMMMMMMM  t5: MMMMMMMM   vl: MM   vs: MMMMMMMM
a6: MMMMMMMM  s6: MMMMMMMM MMMMMMMM  t6: MMMMMMMM vm_u: MMMMMMMM MMMMMMMM
a7: MMMMMMMM  s7: MMMMMMMM MMMMMMMM  t7: MMMMMMMM vm_l: MMMMMMMM MMMMMMMM
global_int_enab: MM  local_int_enab: MM  int_mode: MM  target_CPU: M  ION: M
```

In Figure 119 the following fields have specific meaning:

Field	Description
N	The CPU number.
MMMMMMMM	The current value of the specified register.

Interrupt error messages

The error messages in this section result from the occurrence of an exception on the system.

The error message in Figure 120 usually results from the service processor being unable to read or write to main memory. Possible causes include a confused memory board or EBUS arbitrator, or the system pulled a hard error.

Figure 120
Example error messages for class 5 read/write error

```
probable cause was EBUS window AAAAAAAA
SPU address: 0xBBBBBBBB   MM address 0xCCCCCCCC
EBUS map:     address: 0xHHHHHHHH  bits set: IIIIIIII
MM address translates to:
      slot: JJJ   row: K    bank: L   MMMMMMMM

< address mapping information - see above section >
```

The error message in Figure 121 is displayed if a hard error occurs while running a subtest.

Figure 121
Example error messages for class 5 hard error

```
received unexpected hard error - ESR is 0xEE
< hard logger output >

< address mapping information - see above section >
```

The error message in Figure 122 is displayed if a soft error occurs while running a subtest.

Figure 122
Example error messages for class 5 soft error

```
received unexpected soft error - SEL is 0xEE
< hard logger output >

< address mapping information - see above section >
```

The error message in Figure 123 occurs if the service processor receives an unexpected interrupt from a CPU. Possible causes include the CPU taking an exception and executing a halt instruction, or the occurrence of a miscompare with halt-on-error enabled.

Figure 123
Example error messages for class 5 CPU interrupt

```
< CPU register dump - for failing CPU - see above section >
< a multi-line description of the expected meanings of the registers >

< address mapping information - see above section >
```

In Figure 120, Figure 121, Figure 122, and Figure 123 the following fields have specific meaning:

Field	Description
AAAAAAAA	The EBUS operation that failed.
0xBBBBBBBB	The service processor window address.
0xCCCCCCCC	The main memory address.
D	The CPU number.
0xEE	The value of the indicated register.
0xHHHHHH	The address of the window map register.
IIIIIIII	The operation bits set in the window map register.

Field	Description
JJJ	The memory board the access was made to.
K	The row the access was made to.
L	The bank the access was made to.
MMMMMMMM	The device coordinates the access was made to.
NNNNNNNN	An explanation of the halt code in the A6 register.

Common error messages

The error message in Figure 124 is an indication of a software problem.

Figure 124
Example error messages for class 5 software problem

```
cpu_pat:    unknown subtest number FFFF
```

The error message in Figure 125 indicates the CPU code could not be loaded into main memory. A possible cause is a corrupt executable.

Figure 125
Example error messages for class 5 load error

```
load of CPU code failed (mem_load returned AA)
message: BBBBBBBB
```

The three error messages in Figure 126 indicate a probable problem with the executable which is loaded into main memory.

Figure 126
Example error messages for class 5 code problem

```
unable to find entry point (find_sym returned AA)
```

```
unable to find 'Stack_struct' (find_sym returned AA)
```

```
unable to find 'Stack_end' (find_sym returned AA)
```

The error message in Figure 127 indicates memory mapping was unsuccessful.

Figure 127
Example error messages for class 5 memory mapping

```
unable to map memory for CPU test (map_mem returned AA)
```

The error message in Figure 128 indicates that system initialization did not complete successfully.

Figure 128
Example error messages for class 5 initialization

```
unable to initialize system (init_system returned AA)
< address mapping information - see above section >
```

The error message in Figure 129 indicates that a CPU did not finish in the allotted time, but did not terminate abnormally, either.

Figure 129
Example error messages for class 5 abnormal termination

```
< address mapping information - see above section >

< CPU register dump - for each CPU used - see above > section
```

The error message in Figure 130 indicates the service processor could not read from main memory. Possible causes include a confused memory board or EBUS arbitrator, or the system has pulled a hard error.

Figure 130
Example error messages for class 5 SPU read

```
unable to read communication structures from main memory (rtn = AA)
< address mapping information - see above section >
```

The error message in Figure 131 indicates the service processor could not write to main memory. Possible causes include a confused memory board or EBUS arbitrator, or the system has pulled a hard error.

Figure 131
Example error messages for class 5 SPU write

```
unable to write initial communication structures to main memory (rtn = AA)
< address mapping information - see above section >
```

The three error messages in Figure 132 indicate a default operation failed for some unknown reason.

Figure 132
Example error messages for class 5 operation failed

```
level 2 mcm_default failed - test aborted
```

```
level 0 pia_default failed - test aborted
```

```
level 0 cpx_default failed - test aborted
```

In Figure 132 the following fields have specific meaning:

Field	Description
AA	Return value.
BBBBBBBB	The message string returned by the failing routine.

Subtests 5000-5030—Error messages

The two error messages in Figure 133 are indications of software problems.

Figure 133
Example error messages for subtests 5000-5030 code

```
unable to find any data in 'Util_data' for subtest FFFF
```

```
unable to compute logical address for physical addres 0xCCCCCCCC
< address mapping information - see above section >
```

The three error messages in Figure 134 indicate the service processor could not talk to main memory. Possible causes include a confused memory board or EBUS arbitrator, or the system pulled a hard error.

Figure 134
Example error messages for subtests 5000-5030 memory

```
unable to write command structure to CPU D (rtn = AA)
< address mapping information - see above section >
```

```
unable to read command structures from CPU D (rtn = AA)
< address mapping information - see above section >
```

```
unable to write command code to CPU D (rtn = AA)
< address mapping information - see above section >
```

The three error messages in Figure 135 indicate either a software problem, or that going through main memory to get from a CPU to the service processor (or the other way around), a data value was corrupted.

Figure 135
Example error messages for subtests 5000-5030 memory

```
< address mapping information - see above section >
```

```
< CPU register dump - for failing CPU - see above section >
```

```
unknown op EE
```

The error message in Figure 136 indicates the operation being tested failed.

Figure 136
Example error messages for subtests 5000-5030 operation failed

```
JJJJJJJJ failed to physical address 0xCCCCCCCC
data starting ar physical address 0xKKKKKKKK, logical address
0xLLLLLLLL
actual:     0xMMMM 0xMMMM 0xMMMM 0xMMMM 0xMMMM 0xMMMM 0xMMMM 0xMMMM
expected:   0xNNNN 0xNNNN 0xNNNN 0xNNNN 0xNNNN 0xNNNN 0xNNNN 0xNNNN
< address mapping information - see above section >
```

In Figure 136 the following fields have specific meaning:

Field	Description
AA	Return value.
0xCCCCCCCC	The physical address that caused the problem.
D	The CPU number.
EE	The illegal status value.

HHHHHHHH	The status description, if known (may be omitted). One of:
	`CPU is WAITING for CONTINUE`
	`unknown command`
	`unknown memory utility operation`

II	The unknown operation code.

JJJJJJJJ	The operation that failed, one of:
	`long-word (64-bit) write`
	`word (32-bit) write`
	`half-word (16-bit) write`
	`byte (8-bit) write`
	`TAS operation`
	`TAC operation`

0xKKKKKKKK	The physical address of the beginning of the memory dump.
0xLLLLLLLL	The virtual address of the beginning of the memory dump.
MMMM	The actual data read from memory.
NNNN	The expected data.

Subtests 5100-5120—Error messages

The three error messages in Figure 137 indicate the service processor could not talk to main memory. Possible causes include a confused memory board or EBUS arbitrator, or the system pulled a hard error.

Figure 137
Example error messages for subtests 5100-5120 memory

```
unable to write command structure to main memory on check
< address mapping information - see above section >
```

```
unable to restart CPU D
< address mapping information - see above section >
```

```
unable to continue CPU D after error
< address mapping information - see above section >
```

The error message in Figure 138 indicates that the data at an address was not as expected.

Figure 138
Example error messages for subtests 5100-5120 address

```
miscompare from CPU X
logical address:          0xIIIIIIII  physical address:        0xJJJJJJJJ
actual address (data):    0xKKKKKKKK  expected data (address):0xMMMMMMMM
< address mapping information - see above section >
```

In Figure 138 the following fields have specific meaning:

Field	Description
X	The CPU number.
0xIIIIIIII	The logical address read from.
0xJJJJJJJJ	The physical address read from.
0xCCCCCCCC	The actual data read from the address.
0xMMMMMMMM	The expected data.

The error message in Figure 139 is an indication of a software problem.

Figure 139
Example error messages for subtests 5100-5120 code

```
fatal error - program bug - adr = data test can only have one block per CPU
```

Subtests 5200-5330—Error messages

The three error messages in Figure 140 indicate the service processor could not talk to main memory. Possible causes include a confused memory board or EBUS arbitrator, or the system pulled a hard error.

Figure 140
Example error messages for subtests 5200-5330 memory

```
unable to write data for CPU D restart
< address mapping information - see above section >
```

```
unable to retart CPU D with new block
< address mapping information - see above section >
```

```
unable to continue CPU D after error
< address mapping information - see above section >
```

The error message in Figure 141 indicates that the data at an address was not as expected.

Figure 141
Example error messages for subtests 5200-5330 address

```
miscompare from CPU D
logical address:         0xHHHHHHHH  physical address:      0xIIIIIIII
actual high/odd word:    0xJJJJJJJJ  actual low/even word   0xKKKKKKKK
expected high/odd word:  0xLLLLLLLL  expected low/even word 0xMMMMMMMM
[ location is: AAAAAAAA ]
< address mapping information - see above section >
```

In Figure 141 the following fields have specific meaning:

Field	Description
AAAAAAAA	The location in the algorithm of the failure.
D	The CPU number.
0xHHHHHHHH	The failing longword logical address.
0xIIIIIIII	The failing longword physical address.
0xJJJJJJJJ	The actual upper (address-wise) 32 bits.
0xKKKKKKKK	The actual lower (address-wise) 32 bits.
0xLLLLLLLL	The expected upper (address-wise) 32 bits.
0xMMMMMMMM	The expected lower (address-wise) 32 bits.

PI2 functional tests (pi2_4000)

10

The pi2_4000 test is a functional test for the PBUS interface adapter 2 (PI2). All major functional units of the PI2 are tested, including:

- PBUS interface logic
- RAMs
- Arbitration

In addition, other functionality that does not reside in any one area is also tested (such as error handling).

The following are areas of functionality that cannot be tested adequately:

- Arbitration logic can only be exhaustively tested when a full complement of PI2s is installed. A small amount of functionality is not tested in most configurations
- Paths between the backplane and the scan registers are not testable with scan-based tests alone (use io4000 or other CCU tests to cover this functionality)
- The CCU-clock logic is not testable completely with scan-based tests (use io4000 to cover this functionality)
- The scan logic is also not tested. spu4000 should be used to verify this functionality

This test offers two modes of operation. The first is the traditional "fail /no-fail" testing. The second mode of operation includes the ability to single step (trace) through the entire scan-based portion of the test or up to the point of failure. Included in this mode is the capability to enter iscn or sputil before each step.

When in the single step mode, the test will stop before each scan operation (such as, clock, compare, write to field, and so forth) and display the operation to be executed next. At that point, a shell can be forked so that iscn or sputil can be used to manipulate or examine the board during the diagnostic subtest. Otherwise, press **RETURN** to continue.

The scan-based portion of the test can be modified using a scan-language interface that is described in the *CONVEX Processor Diagnostics Manual (C200 Series) Scan-Language Interface Supplement*. This ability is designed primarily for CONVEX manufacturing and engineering personnel for the purposes of debugging. The tools necessary to utilize the scan-language interface are not shipped to field personnel and customers.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 90.

Table 90
`pi2_4000` functional areas tested

Functional area	Tested by diagnostic	Exercised by the diagnostic
CPU	No	No
CUJ	No	Yes
Memory even	No	Yes
Memory odd	No	Yes
PI2	Yes	No
CCU	No	No
SP5	Yes	No

In order to run the `pi2_4000` test, the boards listed in Table 91 must be operational. The table shows the tests used to verify the required boards.

Note

In order to verify the full functionality, the system under test must contain as many PI2s as allowed in the system. However, only a very minimal portion of functionality is not tested when fewer than possible PI2s are present. A successful test will, in most cases, yield an acceptable level of confidence in the board(s).

Table 91
`pi2_4000` required functional boards

Board	Test to verify
Service processor	`spu1000, spu4000`
CPU utility board (CUJ)	`CUJ_SCAN`
Memory system	`mem4000`

Note

Memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the `pi2_4000` test, use the procedure shown in Figure 142.

Figure 142
`pi2_4000` test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL

: test pi2_4000 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if `mem4100` was executed with failures, or if `spu4000` was executed, then `initall` must be executed prior to test execution. It should be run only after the EPROM based self-test has passed. Failure to execute `initall` in these circumstances could result in invalid test results.

Caution

Do sysreset before attempting to run `pi2_4000`. If sysreset is not done, some of the subtests could erroneously indicate a malfunction.

Entering only

```
test pi2_4000
```

executes all `pi2_4000` subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the `-c` or `-s` options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 143.

Figure 143
`pi2_4000` test parameter menu

```
Test 'pi2_4000'                                  Thu Nov 19 00:00:00 1986

                        ENTER TEST PARAMETERS

             [ ]   Encloses allowed input ranges or values
             ( )   Encloses the default value
             ^     Returns to the previous prompt
             :nn   Returns to the prompt # nn
             :     Returns to the first unsatisfied prompt
             :?    Reviews previous entries

1: Enter Pi2 to test (0=piy,1=pix): [0-1]                        (0) ->
2: Enter value of trace flag: [0-1]                              (0) ->
3: Enter value of extended test flag: [0-1]                      (0) ->
4: Enter OK, or :NN to return to question NN [OK]               (OK) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

Prompt explanations

The following section describes each of the previous prompts.

```
1: Enter PI2 to test (0-piy,1-pix): [0-1]          (0) ->
```

This prompt allows selection of the particular PI2 board to be tested. Enter `0` for piy, and `1` for pix.

```
2: Enter value of trace flag: [0-1]                (0) ->
```

This prompt allows the trace mode to be turned on or off. Enter `0` to turn trace mode off, or enter `1` to turn trace mode on. Refer to the previous explanation of trace mode in the overview of this chapter.

```
3: Enter value of extended test flag: [0-1]        (0) ->
```

This prompt allows extended test mode to be turned on or off. Enter `0` to turn extended mode off, or enter `1` to turn extended mode on. Executing the diagnostic in extended mode requires more time than in regular mode because it causes the diagnostic to perform more extensive tests. In most cases, adequate fault coverage can be achieved without executing in the extended mode.

```
4: Enter OK, or :NN to return to question NN [OK]  (OK) ->
```

If `OK` (default) is selected, test execution begins. A particular prompt can be changed by entering its number here and the program changes to that prompt.

Class description

The PI2 functional subtests are all in one class containing six different types of subtests that each test a logical section of the board. The six types of subtests are associated with the following areas:

- Clock
- PBUS
- RAM
- Transfer
- Error
- Arbitration

In most cases the subtests associated with transfers have a tendency to test all areas working together.

Subtest descriptions

The following subtests are described by the type of function performed by the subtest. If no errors are detected during the test, the response for each subtest is passed. If any of the following subtests fail, the PI2 board(s) must be replaced.

Table 92 describes each of the subtests. The hardware tested by each subtest is mentioned in the "Test performed" column.

Table 92
PI2 functional subtests

Subtest	Test performed
100	Clock alignment
150	Clock state machine
200	PBUS integrity
250	Log ring lock-on-error
300	PBUS parity checker
310	PBUS parity checker
350	PBUS arbitration
400	PBUS illegal header
450	Memory base pointer
500	PCM RAM
550	Write/control queue RAM
600	Write transfer
650	Arbitration queue RAM
700	Return queue RAM
750	EBUS parity checker
800	Read transfer
850	Test-and-modify (TAM) transfer
900	Memory
950	Hard error
1000	Non-present memory
1050	Write data parity error
1150	Interrupt function

Subtest 100—Clock alignment

This subtest verifies that the clock logic on the service processor and the PI2 can be properly aligned and that this alignment is preserved when the rings are clocked. It will test the ability of the 10 MHz ring to be scanned in the following circumstances:

- After sysreset has been issued
- After each of 5 single 10 MHz clocks (single step clocks)
- After groups of 1, 2, 3, 4, and 5 10 MHz clocks (multi-step clocks)

Subtest 150—Clock state machine

This subtest is almost identical to the clock alignment subtest except that instead of checking the service processor's scan-OK bits, in the scan feedback register (SFR), it will look at the internal state of the clock logic via look-a-side scan. The cases tested are as follows:

- After sysreset has been issued
- After each of 5 single 10 MHz clocks (single step clocks)
- After groups of 1, 2, 3, 4, and 5 10 MHz clocks (multi-step clocks)

Subtest 200—PBUS integrity

This subtest is broken into two parts:

- Verification of a properly floating bus on reset
- Pattern testing of the PBUS in loopback mode

The patterns tested are the following:

- 64-bit word of all 0s
- 64-bit word of all Fs
- 64-bit word of all As
- 64-bit word of all 5s
- 0x123456789ABCDEF0

Subtest 250—Log ring lock-on-error

This subtest verifies that the log ring actually "freezes up" in an error condition. Both EBUS and PBUS sections are checked to verify that the lock bits (and only the lock bits) do indeed lock their respective sections.

Subtest 300—PBUS parity checker

This subtest verifies that the parity checker on the PBUS does correctly detect bad parity coming into the PI2 from the PBUS and that the parity checker does not spuriously indicate bad parity. The following cases are examined:

- Verify good parity for data all 0s, parity all Fs
- Walk a 0 through parity for data all 0s
- Walk a 1 through all bytes of the data words with Fs in the parity

Subtest 310—PBUS parity checker

This subtest verifies that the parity checker on the PBUS does correctly detect bad parity coming from the PI2 to the PBUS and that the parity checker does not spuriously indicate bad parity. The parity checker is tested by walking a one across each byte of the longword in the parity checker. The parity is recalculated each time after the pattern is walked through the parity checker's longword. The following cases are examined:

- Bad parity is issued to verify that the parity checker can in fact detect bad parity
- Good parity is issued to verify that spurious bad parity is not generated by the hardware that detects bad parity
- Parity checker is examined to verify that bad parity is not reported when parity is good

Subtest 350—PBUS arbitration

This subtest verifies that single and multiple PBUS requests are properly arbitrated. A request is set up in scan and arbitration logic is clocked to verify that it does indeed arbitrate correctly. Both single request arbitration and multiple request arbitration are verified.

Subtest 400—PBUS illegal header

This subtest checks that the PI2 properly detects incorrectly constructed PBUS headers. The following cases are tested:

- Header bad parity detection
- TAS (test-and-set) transfer with byte count not equal to one
- TAC (test-and-clear) transfer with byte count not equal to one
- I/O write transfer
- Unimplemented PBUS type
- I/O read with illegal address

Subtest 450—Memory base pointer (MBP) read

This subtest verifies the integrity of the MBP data paths using patterns of 0s, Fs, As and 5s on the PBUS.

Subtest 500—PCM RAM

This subtest verifies that all possible patterns (0, 1, 2, 3) can be written and read from the PCM RAM.

Subtest 550—Write and control queue RAM

This subtest verifies that the patterns (0, f, 3, 5, 6, 9, a, c) can be written to and read from all RAM locations.

This subtest consists of two verifications:

- Verify all data values can be written to location 0.
- Verify that data patterns (0 , f, 3, 5, 6, 9, a, c) can be written to all RAM locations.

Subtest 600—Write transfer

Subtest 600 verifies the following:

- Longword aligned single transfer
- Longword aligned dual transfer
- Write abort
- Service processor write
- The zone is calculated correctly for all possible zone combinations
- Burst mode
- Queue full or empty conditions

Subtest 650—Arbitration queue RAM

This subtest consists of two verifications:

- Verify all data values can be written to location 0
- Verify that data patterns (0, f, 3, 5, 6, 9, a, c) can be written to all RAM locations

Subtest 700—Return queue RAM

Subtest 700 verifies that data patterns (0, f, 3, 5, 6, 9, a, c), repeated in all nibbles, can be read and written to the RAM.

Subtest 750—EBUS parity checker

This subtest checks that the parity checker on the EBUS does in fact correctly detect bad parity and does not spuriously indicate bad parity. The following cases are examined:

- Verify good parity for data all 0s, parity all Fs
- Walk a 0 through parity for data all 0s
- Walk a 1 through all bytes of the data words with Fs in the parity

Subtest 800—Read transfer

Subtest 800 verifies the following:

- Single word transfer
- Dual transfer
- Burst mode operation
- Queue full or empty handling
- Transfer abort handling
- Simulated full 64 kbyte transfer

Subtest 850—Test-and-modify (TAM) transfer

Subtest 850 verifies the following:

- Test-and-set (TAS) transfer
- Test-and-clear (TAC) transfer
- Service processor test-and-modify (TAM) transfer

Subtest 900—Memory

This subtest sets up a transfer in scan and lets it finish without checking any machine state. The object is to see if the data is correctly written to and read from memory. The cases tested are as follows:

- Service processor read, service processor write
- PBUS write, service processor read
- Service processor write, PBUS read
- PBUS write, PBUS read

Subtest 950—Hard error

This subtest checks the mechanisms for generating hard errors. It attempts to verify the following:

- PCM parity error detection
- Interrupt arbiter error detection
- Memory data error detection

Subtest 1000—Nonpresent memory (NPM)

The following cases of NPM detection are verified:

- Transfer with starting address in NPM
- Transfer ending in NPM
- Transfer with starting address just after NPM
- Transfer ending at address just before NPM

Subtest 1050—Write data parity error

In this subtest, a transfer is set up and a parity error is forced in each of the following circumstances:

- Error on first write
- Error on last write
- Error on an intermediate write

Subtest 1150—Interrupt function

In this subtest, the interrupt state machine will be verified as well as the grant generation circuitry.

Modes of operation and source files

The `pi2_4000` diagnostic has two modes of execution. The test can be executed in the traditional "fail/no-fail" manner in which the test is executed and if any failures occur, failure messages are output.

The second mode of execution allows the user to step through the test in a tracing mode. This single step method allows the user to step through the test up to the point of failure or to set break points in a failing diagnostic.

The diagnostic itself has two sets of source code: C language source files, and scan-language scripts. The scan-language scripts are compiled together into a single object file which is read into the diagnostic test's data space. This single object file (pi2_4000.x00) is actually a description of threaded code to be executed by the C language routine (pia_4000.t) of the diagnostic.

Scan language test modification

The scan-based portion of the test can be modified using a scan-language interface that is described in the *CONVEX Processor Diagnostics Manual (C200 Series) Scan-Language Interface Supplement*.

This ability is designed primarily for CONVEX manufacturing and engineering personnel for the purposes of debugging. The tools necessary to utilize the scan language interface are not shipped to field personnel and customers.

Subtest error messages

Two basic types of error messages created by `pi2_4000`. The first type of error message is in a standard format. In most cases, this type of error message is associated with a compare function. The standard type of error message is shown in Figure 144.

Figure 144
Standard error message format—`pi2_4000`

```
================================================================
MISMATCH            EXP    ACT    COMMENT
================================================================
<Mismatch_String> XX     YY     <Explanatory_Statement>
================================================
```

In Figure 144:

`<Mismatch_String>` A description of the field or register that did or did not match

`XX` The expected value from the field

`YY` The actual value of the field

`<Explanatory_Statement>` An explanatory statement about the comparison

Note

The above message format is displayed every time a comparison is made. The only time it should be regarded as an error message is when the actual and expected values do not match.

All other `pi2_4000`-specific error messages are unique in nature and are displayed in the following sections.

Subtest 300—Error messages

Subtest 300 displays an error message that contains both a standard comparison format table (shown previously), and the message in Figure 145.

Figure 145
Example error message for subtest 300

```
st_300_2 Pattern= XXXXXXXX: failed
```

In Figure 145:

XXXXXXXX Hexadecimal pattern that caused the failure.

The pattern should be in the form:

`0x01010101, 0x02020202, ... 0x08080808.`

Subtest 350—Error messages

The error messages in Figure 146 can occur in subtest 350.

Figure 146
Example error messages for subtest 350

```
Incorrect PBUS grant single request Requestor: PBUS device YYYY
```

```
Incorrect PBUS grant multiple request Requestors: PBUS devices [<list
of devices chosen from 0-3> ] Last grantee: PBUS device X Grantee:
PBUS device YYYY Expected grantee: PBUS device ZZZZ
```

In Figure 146:

X	0, 1, 2, or 3
YYYY	NONE, 0, 1, 2, or 3
ZZZZ	NONE, 0, 1, 2, or 3

10.0.1 Subtest 400—Error messages

Subtest 400 returns a combination of the standard compare table and information about the address used and parity, as shown in Figure 147.

Figure 147
Example error message for subtest 400

```
Addr: XXXXXXXX parity= YYYYYYYY Byte count: IIIIIIII Parity JJJJJJJJ
```

In Figure 147, `XXXXXXXX`, `YYYYYYYY`, `IIIIIIII`, and `JJJJJJJJ` are hexadecimal numbers.

Subtest 450—Error message

The error message in Figure 148 can occur in subtest 450.

Figure 148
Example error message for subtest 450

```
st_450_0 Pattern= XXXXXXXX
input parity= YYYYYYYY
output parity= ZZZZZZZZ
```

In Figure 148, `XXXXXXXX`, `YYYYYYYY`, and `ZZZZZZZZ` are hexadecimal numerals.

Subtest 500—Error messages

The error messages in Figure 149 can occur in subtest 500.

Figure 149
Example error messages for subtest 500

```
PCM RAM error SEQUENCE: DDDDDDDD RAM addr= XXXXXXXX
Actual value= YYYYYYYY Expected value: ZZZZZZZZ
```

```
PCM RAM RAM addr= XXXXXXXX
Actual value= YYYYYYYY Expected value: ZZZZZZZZ
```

In Figure 148, DDDDDDDD is a decimal numeral, and XXXXXXXX, YYYYYYYY, and ZZZZZZZZ are hexadecimal numerals.

Subtest 550—Error messages

The error message in Figure 150 can occur in subtest 550.

Figure 150
Example error message for subtest 550

```
st_550: Write cntrl q address= XXXXXXXX
```

In Figure 150, XXXXXXXX is an hexadecimal numeral.

Subtest 600—Error message

The error messages in Figure 151 can occur in subtest 600.

Figure 151
Example error messages for subtest 600

```
st_600_2: failed taddr= XXXXXXXX byte count= YYYYYYYY
```

```
st_600_3: failed taddr= XXXXXXXX byte count= YYYYYYYY
```

In Figure 151, XXXXXXXX and YYYYYYYY are hexadecimal numerals.

Subtest 650—Error messages

The error messages in Figure 152 can occur in subtest 650. These error messages are followed by the standard comparison table.

Figure 152
Example error messages for subtest 650

```
pat= XXXXXXXX Addr= YYYYYYYY old_pat= ZZZZZZZZ
```

```
Error in reading back PREVIOUS DATA
```

```
Error in reading back CURRENT DATA
```

In Figure 152, XXXXXXXX, YYYYYYYY, and ZZZZZZZZ are hexadecimal numerals

Subtest 700—Error message

The error message in Figure 153 can occur in subtest 700.

Figure 153
Example error message for subtest 700

```
st_700: addr= XXXXXXXX
```

In Figure 153, XXXXXXXX is an hexadecimal numeral.

Subtest 750—Error message

The error message in Figure 154 can occur in subtest 750.

Figure 154
Example error message for subtest 750

```
st_750_1 Pattern= XXXXXXXX: failed
```

In Figure 154, XXXXXXXX is an hexadecimal numeral.

Subtest 800—Error messages

The error messages in Figure 155 can occur in subtest 800.

Figure 155
Example error messages for subtest 800

```
addr= XXXXXXXX hdr par= YYYYYYYY hdr hi= ZZZZZZZZ hdr lo= IIIIIIII
```

```
byte count= DDDDDDDD (=XXXXXXXX)
```

```
data hi= XXXXXXXX data lo= YYYYYYYY data par= ZZZZZZZ. maddr= IIIIIIII
```

In Figure 155, `DDDDDDDD` is a decimal numeral, and `XXXXXXXX`, `YYYYYYYY`, `ZZZZZZZZ`, and `IIIIIIII` are hexadecimal numerals.

Subtest 850—Error messages

The error messages in Figure 156 can occur in subtest 850.

Figure 156
Example error messages for subtest 850

```
Unexpected value in mem after TAC
Exp: XXXXXXXX Actual: YYYYYYYY
```

```
Dump of entire long word:      XXXXXXXX
```

```
Unexpected TAS return    Exp: XXXXXXXX Actual: YYYYYYYY
```

```
Unexpected TAS set value       Exp: XXXXXXXX Actual: YYYYYYYY
```

```
Unexpected TAC return  Exp: XXXXXXXX Actual: YYYYYYYY
```

```
Exp: XXXXXXXX Actual: YYYYYYYY
```

In Figure 156, XXXXXXXX and YYYYYYYY are hexadecimal numerals.

Subtest 900—Error messages

Any of the error messages in Figure 157 can occur in subtest 900.

Figure 157
Example error messages for subtest 900

```
Data line test: (SPU read) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Data line test: (PBUS write/read) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Data line test: (PBUS read) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Data line test: (SPU read/write) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Address line test: (SPU read) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Address line test: (PBUS write/read) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Address line test: (PBUS read) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

```
Address line test: (SPU read/write) ADDR: XXXXXXXX
      Expected data: hi= IIIIIIII lo= JJJJJJJJ
      Actual data.  hi= KKKKKKKK lo= LLLLLLLL
```

In Figure 157, `XXXXXXXX`, `IIIIIIII`, `JJJJJJJJ`, `KKKKKKKK`, and `LLLLLLLL` are hexadecimal numerals

Subtest 1150—Error message

The error message in Figure 158 occurs in subtest 1150.

Figure 158
Example error message for subtest 1150

```
Exhaustive req/dev_ctr test Requestor: DDDDDDDD Step: EEEEEEEE
State: expected: FFFFFFFF actual: GGGGGGGG
dev_ctr exp: HHHHHHHH (XXXXXXXX) act: IIIIIIII (YYYYYYYY)
```

In Figure 158, `DDDDDDDD`, `EEEEEEEE`, `FFFFFFFF`, `GGGGGGGG`, `HHHHHHHH`, and `IIIIIIII` are decimal numerals, and `XXXXXXXX` and `YYYYYYYY` are hexadecimal numerals.

Scalar building block tests (cpu4030)

11

The cpu4030 test is the most basic of the CPU functional tests and is designed to verify basic functionality by testing all nonprivileged scalar instructions in a nonexhaustive manner.

The cpu4030 test begins verification of the CPU by executing a variety of basic instructions.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 93.

Table 93
`cpu4030` functional areas tested

Functional area	Tested by diagnostic	Exercised by the diagnostic
CPU	Yes	No
CUJ	Yes	No
Memory even	No	Yes
Memory odd	No	Yes
PI2	No	Yes
CCU	No	No
SP5	No	Yes

In order to run the `cpu4030` test, the boards listed in Table 94 must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table 94
`cpu4030` required functional boards

Board	Test to verify
Service processor (SP5)	`spu1000, spu4000`
PBUS interface adapter (PI2)	`pi2_4000`
Memory (mcm, mcm2, mcm3)	`mem4100`
CPU utility board (CUJ)	`CUJ_SCAN`

Note

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the cpu4030 test, use the procedure shown in Figure 159.

Figure 159
cpu4030 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL

: test cpu4030 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if `mem4100` was executed with failures, or if `spu4000` was executed, then `initall` must be executed prior to test execution, but only after the SPU EPROM based self-test has passed. Failure to execute `initall` in these circumstances could result in invalid test results.

Entering only

```
test cpu4030
```

executes all cpu4030 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the -c or -s options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+>*filename*] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 160.

Figure 160
`cpu4030` test parameter menu

```
                ENTER TEST PARAMETERS

      [ ]   Encloses allowed input ranges or values
      ( )   Encloses the default value
      ^     Returns to the previous prompt
      :nn   Returns to the prompt # nn
      :     Returns to the first unsatisfied prompt
      :?    Reviews previous entries

1: Run default switches? [y,n]                          (y) ->
2: CPUs to test: [01234567]                    (01234567) ->
3: Parallel test execution? [y,n]                       (n) ->
4: Forced Faulting Enabled? [y,n]                       (n) ->
5: Fault on Instruction Fetches? [y,n]                  (n) ->
6: Sequential Execution? [y,n]                          (n) ->
7: Timeout Scale Factor Enabled? [1-100]                (1) ->
8: Dcache Enabled? [y,n]                                (y) ->
9: Segment of Execution? [0-7]                          (0) ->
10: Loop Enabled? [y,n]                                 (n) ->
11: Chained Execution Mode? [y,n]                       (n) ->
12: Hard Errors Enabled? [y,n]                          (y) ->
13: Load CPU Code? [y,n]                                (y) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

In Figure 160, prompt 2, `[01234567]` represents all available CPUs in the machine under test.

Prompt explanations

A description of each prompt and its meaning follows.

```
1: Run default switches? [y/n]                (y) ->
```

If you respond with **y** or **RETURN**, no additional test parameter prompts are displayed and testing begins.

However, if you respond with **n**, additional test parameter prompts are displayed, allowing choices other than the default selections. The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
2: CPUs to test: [01234567]              (01234567) ->
```

This prompt allows selection of the CPU(s) to be used in the test. The possible selections, and the default, represented by `01234567`, consist of all available CPUs.

```
3: Parallel test execution?                   (n) ->
```

This prompt allows parallel test execution to be enabled or disabled. This prompt is only displayed if the `CPUs to test:` prompt is answered with multiple CPUs.

```
4: Forced Faulting Enabled? [y,n]             (n) ->
```

If you answer with **y**, normal force faulting occurs on all data references: the system forces a nonresident data exception to occur on every data reference. If this option is enabled, subtest execution time is increased and the timeout scale factor requires adjustment to prevent the service processor from terminating the test prematurely.

```
5: Fault on Instruction Fetches? [y,n]        (n) ->
```

If you answer with **y**, force faulting occurs on instruction fetches, in addition to data references. This prompt appears only if the previous prompt is answered **y**.

```
6: Sequential Execution? [y,n]                (n) ->
```

If you answer with **y**, the sequential bit in the processor status word (PSW) is set to forced sequential execution mode.

```
7: Timeout Scale Factor [1-100]                (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number entered to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
8: Dcache Enabled? [y,n]                       (y) ->
```

The Dcache is normally enabled. However, if it is suspected to be broken, it can be disabled by entering **n** at this prompt.

```
9: Segment of Execution? [0-7]                 (0) ->
```

This prompt is only displayed if forced faults are not enabled. The segment of execution is contained in bits <31..29> of the program counter (PC):

- If **0** is entered, then bits <31..29> of the PC are `000` and the test is run in ring zero.
- If **1** is entered, then bits <31..29> of the PC are `001`, and the test is run in ring one.
- If **2** is entered, then bits <31..29> of the PC are `010`, and the test is run in ring two.
- If **3** is entered, then bits <31..29> of the PC are `011`, and the test is run in ring three.
- If **4, 5, 6,** or **7** is entered, then bits <31..29> of the PC are `100, 101, 110,` and `111` respectively, and the test is run in ring four.

Refer to the *CONVEX Architecture Reference Manual (C Series)* for more information concerning the meaning of rings in the machine architecture.

```
10: Loop Enabled? [y,n]                        (n) ->
```

If you answer with **y**, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by pressing **CTRL-C**.

```
11: Chained Execution Mode? [y,n]          (y) ->
```

With this option enabled, the test is executed in chained mode, which causes the CPU to perform subtest sequencing. The service processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time is greatly reduced.

The only information printed to the console upon completion or failure (regardless of the cause) is the message, `Subtest 1 passed`, or `Subtest 1 failed`. This option cannot be enabled if the `-c` or the `-s` options were used in the invocation procedure.

Note

This option disables all class 4 subtests.

```
12: Hard Errors Enabled? [y,n]             (y) ->
```

If this option is enabled by entering **y**, and a hard error occurs, the clocks are stopped and the test fails. If this option is disabled by entering **n**, parity errors and other sources of hard errors go undetected. It is recommended that hard errors normally be enabled.

```
13: Load CPU Code? [y,n]                   (y) ->
```

If the CPU code for this test is already in memory, enter **n** for this prompt and the code is not reloaded (thus saving time).

Test parameter summary

When all prompts have been answered, the screen displays a test parameter summary which echoes the prompts that have been answered, as illustrated in Figure 161.

Figure 161
cpu4030 test parameter summary

```
                Test Parameter Summary

Run default switches?                 : y
CPUs to test:                         : 01
Parallel test execution?              : n
Forced Faulting Enabled?              : y
Sequential Execution?                 : n
Timeout Scale Factor Enabled?         : 1
Dcache Enabled?                       : y
Segment of Execution?                 : 0
Loop Enabled?                         : n
Hard Errors Enabled?                  : y
Load CPU Code?                        : y
```

The actual summary varies depending on the answers to the prompts.

Hardware initialization sequence

After the last prompt is entered, and before test code execution, the following events occur:

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact virtual to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CUJ, PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- Clocks are turned on to the processor(s) selected. If parallel execution is selected, all clocks are turned on. If sequential execution is selected, each CPU's clocks are turned on one at a time, in the order the CPUs are selected.

Note

The first two events are accomplished at the initial start of the `cpu4030` test. The remaining events are accomplished when each subtest is initialized.

Current memory allocation

Immediately before test code execution, a current memory allocation screen is displayed. Figure 162 is an example of the current memory allocation screen.

Figure 162
cpu4030 current memory allocation screen

```
                    Current Memory Allocation

File No. Physical Address Pid File Name                Logical Offset
         00000000-00027fff  0 p0r0_4030                00000000
         00028000-000b1fff  0 cpu4030.rnn               00022000
         000b2000-000d9fff  1 p0r0_4030                00000000
         000da000-00163fff  1 cpu4030.rnn               00022000
         03ff7000-03ff9fff  1 ptet                      NA
         03ffa000-03ffafff  1 pte2                      NA
         03ffb000-03ffdfff  0 ptet                      NA
         03ffe000-03ffefff  0 pte2                      NA
         03fffc00-3fffffff  1 pte1                      NA
```

The physical and logical addresses shown, as well as the file names, are only representative. The actual addresses will vary depending on installed memory and file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First column**—File number
- **Second column**—Physical memory addresses where the specified file is loaded (Useful in conjunction with the mm(1d) utility).
- **Third column**—Process identification
- **Fourth column**—File name. The actual path is */filename* or /mnt/test/CPU/*filename*. The entries `pte1`, `pte2`, and `ptet` are not actual files, but are indications of the page tables.
- **Fifth column**—Logical (virtual) starting address of the specified file.

Subtest descriptions

There are four classes of subtests for `cpu4030`.

In the following tables, the "Instruction tested/test performed" column lists each instruction that is tested. For more information on individual instructions, refer to the *CONVEX C Series Assembly Language Reference Manual*. The object module for Classes 1, 2, and 3 is cpu4030.rnn.

Class 1 subtests

Class 1 subtests (see Table 95) are the most basic of the CPU verification tests.

Data operands are supplied as immediate data accompanying the instructions. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, the subtest passed. Any other code in A1 indicates the subtest failed.

Table 95
cpu4030 class 1 subtests

Subtest	Instruction tested/test performed
1	`ld.h #N,Ak` `ld.w #N,Ak`
2	`ld.w #N,Sk` `ld.w #N,Ak`
5	`mov Aj,Ak` `mov Aj,Sk`

6	`mov Sj,Ak`
	`mov.w Sj,Sk`
	`mov.l Sj,Sk`

7	`mov PC,Ak`
10	`and Aj,Ak` `and #N,Ak`
11	`and Sj,Sk` `and #N,Sk`
14	`or Aj,Ak` `or #N,Ak`
15	`or Sj,Sk` `or #N,Sk`
16	`mov Ak,psw` `mov psw,Ak`
18	`xor Aj,Ak` `xor #N,Ak`
19	`xor Sj,Sk` `xor #N,Sk`
20	`not Aj,Ak`

Table 95 (continued)
cpu4030 class 1 subtests

Subtest	Instruction tested/test performed
21	`not Sj,Sk`

24	`add.h Aj,Ak`
	`add.w Aj,Ak`
	`add.h #N,Ak`
	`add.w #N,Ak`

25	`add.b Sj,Sk`
26	`add.h #N,Sk` `add.h Sj,Sk`

27	`add.w #N,Sk`
	`add.w Sj,Sk`
	`add.w Sj,Ak`

28	`add.l Sj,Sk`
30	`sub.h #n,a` `sub.h ai,Aj`
31	`sub.w #n,Ak` `sub.w Aj,Ak`
32	`sub.b Sj,Sk`
33	`sub.h #n,Sj` `sub.w Sj,Sk`
34	`sub.w #,Sk` `sub.w Sj,Sk`
35	`sub.l Sj,Sk`
40	`neq.h Aj,Ak`
41	`neq.w Aj,Ak`
42	`neg.b Sj,Sk`
43	`neg.h Sj,Sk`
44	`neg.w Sj,Sk`
45	`neg.l Sj,Sk`
50	`branches`
52	`cmp.h Aj,Ak`
53	`cmp.w Sj,Ak`
54	`cmp.h #N,Ak`
55	`cmp.w #,Ak`

Table 95 (continued)
cpu4030 class 1 subtests

Subtest	Instruction tested/test performed
60	`cmp.b Sj,Sk`
61	`cmp.h Sj,Sk`
62	`cmp.w Sj,Sk`
63	`cmp.h #,Sk`
64	`cmp.w #N,Sk`
65	`cmp.l Sj,Sk`

66	`shf Aj,Ak`
	`shf #n,Ak`
	`shf #N,Ak`

67	`shf Sj,Sk` `shf #N,Sk`
68	`tzc Sj,Sk`
69	`plc.t Sj,Sk`

70	`mul.h #n,Ak`
	`mul.h #N,Ak`
	`mul.h Aj,Ak`

71	`mul.w #n,Ak`
	`mul.w #N,Ak`
	`mul.w Aj,Ak`

72	`mul.b Sj,Sk`
73	`mul.h #N,Sk` `mul.h Sj,Sk`
74	`mul.w #N,Sk` `mul.w Sj,Sk`
75	`mul.l Sj,Sk`

80	`div.h #n,Ak`
	`div.h #N,Ak`
	`div.h Aj,Ak`

81	`div.w #n,Ak`
	`div.w #N,Ak`
	`div.w Aj,Ak`

82	`div.b Sj,Sk`

Table 95 (continued)
cpu4030 class 1 subtests

Subtest	Instruction tested/test performed

83	`div.h #n,Sk`
	`div.h #N,Sk`
	`div.h Sj,Sk`

84	`div.w #n,Sk`
	`div.h #N,Sk`
	`div.h Sj,Sk`

85	`div.l Sj,Sk`
90	`cvt(w.b,b.w,w.h,h.w) Aj,Ak`
91	`cvt(w.b,b.w,w.h,h.w,w.l,l.w) Sj,Sk`

100	`eq.s #n,Sk`
	`eq.s #N,Sk`
	`eq.s Sj,Sk`

101	`le.s #n,Sk`
	`le.s #N,Sk`
	`eq.s Sj,Sk`

102	`lt.s #n,Sk`
	`lt.s #N,Sk`
	`eq.s Sj,Sk`

103	`eq.d Sj,Sk`
104	`le.d Sj,Sk`
105	`lt.d Sj,Sk`
110	`add.s #N,Sk Sj,Sk`
111	`add.d Sj,Sk`
115	`sub.s #N,Sk Sj,Sk`
116	`sub.d Sj,Sk`
125	`neg.s Sj,Sk`
126	`neg.d Sj,Sk`
130	`mul.s #N,Sk` `mul.s Sj,Sk`
131	`mul.d Sj,Sk`
132	`div.s #N,Sk` `div.s Sj,Sk`
133	`div.d Sj,Sk`

Table 95 (continued)
cpu4030 class 1 subtests

Subtest	Instruction tested/test performed
140	`cvtw.s Sj,Sk`
142	`cvts.w Sj,Sk`
144	`cvtd.l Sj,Sk`
146	`cvtl.d Sj,Sk`
148	`vtl.s Sj,Sk`
150	`cvts.l Sj,Sk`
152	`cvts.d Sj,Sk`
154	`cvtd.s Sj,Sk`
1100	`eq.s #N sk eq.s sj sk` (IEEE mode)
1101	`le.s #N sk le.s sj sk` (IEEE mode)
1102	`lt.s #N sk lt.s sj sk` (IEEE mode)
1103	`eq.d sj sk` (IEEE mode)
1104	`le.d sj sk` (IEEE mode)
1105	`lt.d sj sk` (IEEE mode)
1110	`add.s #N sk add.s sj sk` (IEEE mode)
1111	`add.d sj sk` (IEEE mode)
1115	`sub.s sj sk sub.s #N sk` (IEEE mode)
1116	`sub.d sj sk` (IEEE mode)
1127	`neg.s sj sk` (IEEE mode)
1128	`neg.d sj sk` (IEEE mode)
1131	`mul.d sj sk` (IEEE mode)
1132	`div.s #n sk div.s sj sk` (IEEE mode)
1133	`div.d sj sk` (IEEE mode)
1135	`mul.s #N sk mul.s sj sk` (IEEE mode)
1140	`cvtw.s sj sk` (IEEE mode)
1142	`cvtw.s sj sk` (IEEE mode)
1144	`cvtd.l sj sk` (IEEE mode)
1146	`cvtl.d sj sk` (IEEE mode)

Table 95 (continued)
cpu4030 class 1 subtests

Subtest	Instruction tested/test performed
1148	`cvtl.s sj sk` (IEEE mode)
1150	`cvts.l sj sk` (IEEE mode)
1152	`cvts.d sj sk` (IEEE mode)
1154	`cvtd.s sj sk` (IEEE mode)

Class 2 subtests

Class 2 subtests (see Table 96) terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 96
cpu4030 class 2 subtests

Subtest	Instruction tested/test performed
200	`ld.b a,Ak` byte boundaries
201	`ld.h a,Ak` halfword boundaries
202	`ld.w a,Ak` word boundaries
205	`ld.b a,Sk` byte boundaries
206	`ld.h a,Sk` halfword boundaries
207	`ld.w a,Sk` word boundaries
208	`ld.l a,Sk` word boundaries
211	`ld.h a,Ak` unaligned
212	`ld.w a,Ak` unaligned
216	`ld.h a,Sk` unaligned
217	`ld.w a,Sk` unaligned
218	`ld.l a,Sk` unaligned
220	`st.b Ak,a` byte boundaries
221	`st.h Ak,a` halfword boundaries
222	`st.w Ak,a` word boundaries
225	`st.b Sk,a` byte boundaries
226	`st.h Sk,a` halfword boundaries
227	`st.w Sk,a` word boundaries
228	`st.l Sk,a` word boundaries
231	`st.h Ak,a` unaligned boundaries
232	`st.w Ak,a` unaligned boundaries
236	`st.h Sk,a` unaligned boundaries

Table 96 (continued)
cpu4030 class 2 subtests

Subtest	Instruction tested/test performed
237	`st.w Sk,a` unaligned boundaries
238	`st.l Sk,a` unaligned boundaries

240	`psh.l Sk psh.w Sk psh.w Ak`
	`pop.w Ak pop.l Sk pop.w Sk`
	`pshea`

241	`psh.l Sk psh.w Sk psh.w Ak`
	`pop.w Ak pop.w Sk pop.l Sk`
	`pshea` unaligned

242	`tas a`
245	`ldea a,Ak`
250	`callq` aligned stack `rtnq` aligned stack
251	`callq` unaligned stack `rtnq` unaligned stack
252	`call` aligned stack `rtn` aligned stack
253	`call` unaligned stack `rtn` unaligned stack
254	`calls` aligned stack `rtn` aligned stack
255	`calls` unaligned stack `rtn` unaligned stack
260	`st.b Ak,@a` byte boundaries
261	`st.h Ak,@a` halfword boundaries
262	`st.w Ak,@a` word boundaries
265	`st.b Sk,@a` byte boundaries
266	`st.h Sk,@a` halfword boundaries
267	`st.w Sk,@a` word boundaries
268	`st.l Sk,@a` word boundaries
271	`st.h Ak,@a` unaligned boundaries
272	`st.w Ak,@a` unaligned boundaries
276	`st.h Sk,@a` unaligned boundaries

Table 96 (continued)
cpu4030 class 2 subtests

Subtest	Instruction tested/test performed
277	`st.w Sk,@a` unaligned boundaries
278	`st.l Sk,@a` unaligned boundaries
280	`ld.b @a,Ak` byte boundaries
281	`ld.h @a,Ak` halfword boundaries
282	`ld.w @a,Ak` word boundaries
285	`ld.b @a,Sk` byte boundaries
286	`ld.h @a,Sk` halfword boundaries
287	`ld.w @a,Sk` word boundaries
288	`ld.l @a,Sk` word boundaries
291	`ld.h @a,Ak` unaligned
292	`ld.w @a,Ak` unaligned
296	`ld.h @a,Sk` unaligned
297	`ld.w @a,Sk` unaligned
298	`ld.l @a,Sk` unaligned

Class 3 subtests

Class 3 subtests (see Table 97)terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 97
cpu4030 class 3 subtests

Subtest	Test performed
300	Load byte across page boundary
301	Load halfword across page boundary
302	Load word across page boundary
303	Load longword across page boundary
304	Store byte across page boundary
305	Store halfword across page boundary
306	Store word across page boundary
307	Store longword across page boundary
308	Load byte @, address at page boundary
309	Load halfword @, address at page boundary
310	Load word @, address at page boundary
311	Load longword @, address at page boundary
312	Load byte @, address and data at page boundary
313	Load halfword @, address and data at page boundary
314	Load word @, address and data at page boundary
315	Load longword @, address and data at page boundary
350	Execute at page boundary
351	Execute different size op codes at page boundary
352	Execute different size op codes at page boundary
353	Branch near page boundary

Class 4 subtests

Class 4 subtests verify two basic capabilities of the machine that are not explicitly tested anywhere else. The first is the capability to execute code, branch forward and backward across all major carry addresses in the program counter (PC). The second is the verification of the ability to correctly wrap instructions within the current ring.

Carry tests

The carry tests (see Table 98) are performed by subtests 500-515. Each of these subtests use the same object module, however, the module is loaded into a different logical address for each subtest. Each module uses up two pages of logical address space. The logical addresses used are `0xF000`, `0xFF000`, `0xFFF000`, and `0xFFFF000`.

Each subtest executes code across all of the carries possible within its logical address space. For example, subtest 500 will test carry from `0xE` to `0x10`, `0xFE` to `0x100`, `0xFFE` to `0x1000`, and `0xFFFE` to `0x10000`.

Table 98
cpu4030 class 4 carry subtests

Subtest	Test performed
500	PC carry test I
505	PC carry test II
510	PC carry test III
515	PC carry test IV

Ring wrap tests

The ring wrap tests (see Table 99) are performed by subtests 520-555. The ring wrapping is tested by executing code at the end of each segment.

For segments that exist in rings 0, 1, 2, or 3, execution at the end of the segment should cause the PC to "wrap" back down to the beginning of the segment. This prevents violations of the architectural ring protection constraints.

Segments that exist in ring 4, 5, 6, or 7 behave differently. These segments should behave as one ring. For example, it should be possible to execute code from segment 4 and end up in segment 5, segment 5 to segment 6 and segment 6 to segment 7. If code is executed at the end of segment 7, it should go back to the beginning of the ring and end up in segment 4.

Table 99
cpu4030 class 4 ring wrap subtests

Subtest	Test performed
520	PC wraparound test I
525	PC wraparound test II
530	PC wraparound test III
535	PC wraparound test IV
540	PC wraparound test V
545	PC wraparound test VI
550	PC wraparound test VII
555	PC wraparound test VIII

Vector instruction tests (cpu4041)

12

The cpu4041 test is a group of vector instruction tests used to verify the operation of the vector unit and its interfaces to the other CPU subsystems.

The verification is performed by exercising each vector/vector and scalar/vector instruction while varying all of the parameters on which the instructions depend.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 100.

Table 100
`cpu4041` functional areas tested

Functional area	Tested by diagnostic	Exercised by diagnostic
CPU	Yes	No
CUJ	Yes	No
Memory even	No	Yes
Memory odd	No	Yes
PI2	No	Yes
CCU	No	No
SP5	No	Yes

In order to run the `cpu4041` test, the boards listed in Table 101 must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table 101
`cpu4041` required functional boards

Board	Test to verify
Service processor (SP5)	`spu1000, spu4000`
PBUS interface adapter (PI2)	`pi2_4000`
Memory system(mcm, mcm2, mcm3)	`mem4100`
Central processor unit (CPU)	`cpu4030`
CPU utility board (CUJ)	`CUJ_SCAN`

Note

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the cpu4041 test, use the procedure shown in Figure 163.

Figure 163
cpu4041 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL
: test cpu4040 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if mem4100 was executed with failures, or if spu4000 was executed, then initall must be executed prior to test execution, but only after the SPU EPROM based self-test has passed. Failure to execute initall in these circumstances could result in invalid test results.

Entering only

```
test cpu4041
```

executes all cpu4041 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the -c or -s options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 164.

Figure 164
`cpu4041` test parameter menu

```
Test 'cpu4041.t'                              Thu Nov 19 00:00:00 1965 2

                    ENTER TEST PARAMETERS
       [ ]    Encloses allowed input ranges or values
       ( )    Encloses the default value
       ^      Returns to the previous prompt
       :nn    Returns to the prompt # nn
       :      Returns to the first unsatisfied prompt
       :?     Reviews previous entries

1: Run default switches? [y,n]                                  (y) ->
2: CPUs to test: [01234567]                            (01234567) ->
3: Parallel Test Execution? [y,n]                               (n) ->
4: Forced Faulting Enabled? [y,n]                               (n) ->
5: Fault on Instruction Fetches? [y,n.                          (n) ->
6: Sequential Execution? [y,n]                                  (n) ->
7: Timeout Scale Factor Enabled? [1-100]                        (1) ->
8: Dcache Enabled? [y,n]                                        (y) ->
9: Segment of Execution? [0-7]                                  (0) ->
10: Loop Enabled? [y,n]                                         (n) ->
11: Chained Execution Mode? [y,n]                               (n) ->
12: Hard Errors Enabled? [y,n]                                  (y) ->
13: Number of vl values to test(0..vl)? [0-128]               (128) ->
14: Load CPU Code? [y,n]                                        (y) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

In Figure 164, prompt 2, `[01234567]` represents all available CPUs in the machine under test.

Prompt explanations

A description of each prompt and its meaning follows:

```
1: Run default switches? [y/n]                (y) ->
```

If you response with **y** or **RETURN**, no additional test parameter prompts are displayed, and testing begins.

However, if you respond with **n**, additional test parameter prompts are displayed, allowing modification of the default selections. The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
2: CPUs to test: [01234567]              (01234567)->
```

This prompt allows selection of the CPU(s) to be used in the test. The possible selections, and the default, represented by `01234567`, consist of all available CPUs.

```
3: Parallel Test Execution? [y,n]             (n) ->
```

This prompt allows parallel test execution to be enabled or disabled. This prompt is only displayed if the `CPUs to test:` prompt is answered with multiple CPUs.

```
4: Forced Faulting Enabled? [y,n]             (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system forces a nonresident data exception to occur on every data reference.

If this option is enabled, subtest execution time is increased and the timeout scale factor requires adjustment to prevent the SPU from terminating the test prematurely.

```
5: Fault on Instruction Fetches? [y,n]        (n) ->
```

If answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is displayed only if the previous prompt is answered with **y**.

```
6: Sequential Execution? [y,n]                (n) ->
```

If set by entering **y**, the sequential bit in the processor status word (PSW) is set to forced sequential execution mode.

```
7: Timeout Scale Factor Enabled [1-100]    (1)->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it takes the test five times as long to timeout.

```
8: Dcache Enabled? [y,n]                   (y) ->
```

The Dcache is normally enabled. However, if it is suspected to be broken, it may be disabled by entering **n** at this prompt.

```
9: Segment of Execution? [0-7]             (0) ->
```

This prompt is only displayed if forced faulting is not enabled. The segment of execution is contained in bits <31..29> of the program counter (PC).

- If **0** is entered, then bits <31..29> of the PC are `000` and the test is run in ring zero.
- If **1** is entered, then bits <31..29> of the PC are `001` and the test is run in ring one.
- If **2** is entered, then bits <31..29> of the PC are `010` and the test is run in ring two.
- If **3** is entered, then bits <31..29> of the PC are `011` and the test is run in ring three.
- If **4, 5, 6,** or **7** is entered, then bits <31..29> of the PC are `100, 101, 110,` and `111`, respectively, and the test is run in ring four.

Refer to the *CONVEX Architecture Reference Manual (C Series)* for more information about the meaning of rings in the machine architecture.

```
10: Loop Enabled? [y,n]                    (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to branch back to the beginning of the subtest. This puts the subtest into an infinite loop which can be interrupted by entering **CTRL-C**.

```
11: Chained Execution Mode? [y,n]          (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The service processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time will be greatly reduced.

However, the only information printed to the console upon completion or failure (regardless of the cause) is the message, `Subtest 1 passed`, or `Subtest 1 failed`. Also, this option cannot be enabled if the `-c` or the `-s` options were used in the invocation procedure.

```
12: Hard Errors Enabled? [y,n]             (y) ->
```

If this option is enabled and a parity error occurs, the clocks are stopped, and the test fails. If this option is disabled, parity errors and other sources of hard errors go undetected. It is recommended that hard errors normally be enabled.

```
13: Number of vl values to test (0..vl)? [0..128] (128)->
```

Each instruction in this test is executed twice, first with a vector length (VL) value of 128, and second with a VL value ranging from 0 to the VL count. The VL count will be 0 through 128 if the default of this prompt is selected. However, if time is an issue, enable forced faulting and select a VL count of 16 in response to this prompt. This allows decreased subtest execution times without a significant decrease in test coverage.

```
14: Load CPU Code? [y,n]                   (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code is not reloaded, thus saving time.

Test parameter summary

When all prompts have been answered, the screen displays a test parameter summary which echoes the prompts that have been answered, as illustrated in Figure 165.

Figure 165
cpu4041 test parameter summary

```
                TEST PARAMETER SUMMARY

Run default switches?                          : n
Cpus to test:                                  : 01
Parallel Test Execution?                       : n
Forced Faulting Enabled?                       : y
Fault on Instruction Fetches?                  : y
Sequential Execution?                          : y
Timeout Scale Factor Enabled?                  : 1
Dcache Enabled?                                : y
Segment of Execution?                          : 0
Chained Execution Mode?                        : n
Loop Enabled?                                  : n
Hard Errors Enabled?                           : y
Number of vl values to test(0..vl)?            : 128
Load CPU Code?                                 : y
```

The actual summary varies depending on the answers to the prompts.

Hardware initialization sequence

After the last prompt is entered by the user (and before test code execution) the following events occur:

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical-to-physical mapping described by the page tables. The exact logical-to-physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CU, PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- Clocks are turned on to the processor selected. If parallel execution is selected, each CPU's clocks are turned on at one time. If sequential execution is selected, each CPU's clocks are turned on one at a time.

Note

The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

Current memory allocation

Immediately before test code execution, a current memory allocation screen is displayed. Figure 166 is an example of the current memory allocation screen.

Figure 166
cpu4041 current memory allocation screen

```
                    Current Memory Allocation

File No.  Physica. Address  Pid  File Name              Logical Offset
      1   00000000-00027fff   0  p0r0_4041              00000000
      2   00028000-000b1fff   0  cpu4041.rnn            00022000
      1   000b2000-000d9fff   1  p0r0_4041              00000000
      2   000da000-00163fff   1  cpu4041.rnn            00022000
          03ff7000-03ff9fff   1  ptet                   NA
          03ffa000-03ffafff   1  pte2                   NA
          03ffb000-03ffdfff   0  ptet                   NA
          03ffe000-03ffefff   0  pte2                   NA
          03fffc00-3fffffff   1  pte1                   NA
```

The physical and logical addresses shown, as well as the file names, are only representative; the actual addresses will vary depending on installed memory and file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First column**—File number.
- **Second column**—Physical memory addresses where the specified file is loaded (Useful in conjunction with the mm(1d) utility).
- **Third column**—Process identification.
- **Fourth column**—File name. (The actual path is */filename* or /mnt/test/CPU/*filename*. The entries `pte1`, `pte2`, and `ptet` are not actual files, but are indications of the page tables.)
- **Fifth column**—Logical starting address of the specified file.

Subtest descriptions

There are four classes of subtests for `cpu4041`.

In the following tables, the "Instruction tested/test performed" column lists each instruction that is tested. For more information on individual instructions, refer to the *CONVEX C Series Assembly Language Reference Manual*. The object module (the executable code) for all classes is `cpu4041.rnn`.

Class 1 subtests

Class 1 subtests, listed in Table 102, verify the operation of loading, storing, and modifying the vector unit control functions. Specifically, this class verifies the ability to alter and save the vector length (VL) register, vector stride (VS) register, and the vector merge (VM) register.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 102
cpu4041 class 1 subtests

Subtest	Instruction tested/test performed
10	`ld.w #N, VL`
15	`ld.w #N, VS`
20	`mov.w Sk, VL`
25	`mov.w Sk, VS`
30	`mov Ak, VL`
35	`mov VL, Ak`
40	`mov Ak, VS`
45	`mov VS, Ak`
50	`ld.l <effa>, VLS`
51	`ld.l <effa>, VLS`
55	`st.l VLS, <effa>`
56	`st.l VLS, <effa>`
60	`mov Sj, Sk, VM`
65	`mov Sj, VM, Sk`
70	`ld.x <effa>, VM`
75	`st.x VM, <effa>`
80	`plc.t VM, Sk`
85	`plc.f VM, Sk`
195	`mov Si, Sj, Vk`
198	`mov Vi, Sj, Sk`

Table 102 (continued)
cpu4041 class 1 subtests

Subtest	Instruction tested/test performed
1000	Vector valid traps
1010	Dual vector loads

Class 2 subtests

Class 2 subtests, listed in Table 103, verify the operation of the logical and arithmetic pipes. Specifically, this class verifies vector/vector and scalar/vector comparisons, vector/vector and scalar/vector additions and subtractions, and vector reductions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 103
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
105	`le.b Sj, Vk`
106	`le.b Sj, Vk`
110	`le.h Sj, Vk`
111	`le.h Sj, Vk`
115	`le.w Sj, Vk`
116	`le.w Sj, Vk`
120	`le.l Sj, Vk`
121	`le.l Sj, Vk`
125	`le.s Sj, Vk` (native mode)
126	`le.s Sj, Vk` (native mode)
130	`le.d Sj, Vk` (native mode)
131	`le.d Sj, Vk` (native mode)
135	`lt.b Sj, Vk`
136	`lt.b Sj, Vk`
140	`lt.h Sj, Vk`
141	`lt.h Sj, Vk`
145	`lt.w Sj, Vk`
146	`lt.w Sj, Vk`
150	`lt.l Sj, Vk`
151	`lt.l Sj, Vk`

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
155	`lt.s Sj, Vk` (native mode)
156	`lt.s Sj, Vk` (native mode)
160	`lt.d Sj, Vk` (native mode)
161	`lt.d Sj, Vk` (native mode)
165	`eq.b Sj, Vk`
166	`eq.b Sj, Vk`
170	`eq.h Sj, Vk`
171	`eq.h Sj, Vk`
175	`eq.w Sj, Vk`
176	`eq.w Sj, Vk`
180	`eq.l Sj, Vk`
181	`eq.l Sj, Vk`
185	`eq.s Sj, Vk` (native mode)
186	`eq.s Sj, Vk` (native mode)
190	`eq.d Sj, Vk` (native mode)
191	`eq.d Sj, Vk` (native mode)
250	`add.b Vi, Sj, Vk`
251	`add.b Vi, Sj, Vk`
255	`add.h Vi, Sj, Vk`
256	`add.h Vi, Sj, Vk`
260	`add.w Vi, Sj, Vk`
261	`add.w Vi, Sj, Vk`
265	`add.l Vi, Sj, Vk`
266	`add.l Vi, Sj, Vk`
270	`add.s Vi, Sj, Vk` (native mode)
271	`add.s Vi, Sj, Vk` (native mode)
275	`add.d Vi, Sj, Vk` (native mode)
276	`add.d Vi, Sj, Vk` (native mode)

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
280	`sub.b Vi, Sj, Vk`
281	`sub.b Vi, Sj, Vk`
285	`sub.h Vi, Sj, Vk`
286	`sub.h Vi, Sj, Vk`
290	`sub.w Vi, Sj, Vk`
291	`sub.w Vi, Sj, Vk`
295	`sub.l Vi, Sj, Vk`
296	`sub.l Vi, Sj, Vk`
300	`sub.s Vi, Sj, Vk` (native mode)
301	`sub.s Vi, Sj, Vk` (native mode)
305	`sub.d Vi, Sj, Vk` (native mode)
306	`sub.d Vi, Sj, Vk` (native mode)
310	`and Vi, Sj, Vk`
311	`and Vi, Sj, Vk`
315	`or Vi, Sj, Vk`
316	`or Vi, Sj, Vk`
320	`xor Vi, Sj, Vk`
321	`xor Vi, Sj, Vk`
325	`shf Sj, Vk`
326	`shf Sj, Vk`
330	`le.b Vj, Vk`
331	`le.b Vj, Vk`
335	`le.h Vj, Vk`
336	`le.h Vj, Vk`
340	`le.w Vj, Vk`
341	`le.w Vj, Vk`
345	`le.l Vj, Vk`
346	`le.l Vj, Vk`

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
350	`le.s Vj, Vk` (native mode)
351	`le.s Vj, Vk` (native mode)
355	`le.d Vj, Vk` (native mode)
356	`le.d Vj, Vk` (native mode)
360	`lt.b Vj, Vk`
361	`lt.b Vj, Vk`
365	`lt.h Vj, Vk`
366	`lt.h Vj, Vk`
370	`lt.w Vj, Vk`
371	`lt.w Vj, Vk`
375	`lt.l Vj, Vk`
376	`lt.l Vj, Vk`
380	`lt.s Vj, Vk` (native mode)
381	`lt.s Vj, Vk` (native mode)
385	`lt.d Vj, Vk` (native mode)
386	`lt.d Vj, Vk` (native mode)
390	`eq.b Vj, Vk`
391	`eq.b Vj, Vk`
395	`eq.h Vj, Vk`
396	`eq.h Vj, Vk`
400	`eq.w Vj, Vk`
401	`eq.w Vj, Vk`
405	`eq.l Vj, Vk`
406	`eq.l Vj, Vk`
410	`eq.s Vj, Vk` (native mode)
411	`eq.s Vj, Vk` (native mode)
415	`eq.d Vj, Vk` (native mode)
416	`eq.d Vj, Vk` (native mode)

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
430	`add.b Vi, Vj, Vk`
431	`add.b Vi, Vj, Vk`
435	`add.h Vi, Vj, Vk`
436	`add.h Vi, Vj, Vk`
440	`add.w Vi, Vj, Vk`
441	`add.w Vi, Vj, Vk`
445	`add.l Vi, Vj, Vk`
446	`add.l Vi, Vj, Vk`
450	`add.s Vi, Vj, Vk` (native mode)
451	`add.s Vi, Vj, Vk` (native mode)
455	`add.d Vi, Vj, Vk` (native mode)
456	`add.d Vi, Vj, Vk` (native mode)
460	`sub.b Vi, Vj, Vk`
461	`sub.b Vi, Vj, Vk`
465	`sub.h Vi, Vj, Vk`
466	`sub.h Vi, Vj, Vk`
470	`sub.w Vi, Vj, Vk`
471	`sub.w Vi, Vj, Vk`
475	`sub.l Vi, Vj, Vk`
476	`sub.l Vi, Vj, Vk`
480	`sub.s Vi, Vj, Vk` (native mode)
481	`sub.s Vi, Vj, Vk` (native mode)
485	`sub.d Vi, Vj, Vk` (native mode)
486	`sub.d Vi, Vj, Vk` (native mode)
490	`and Vi, Vj, Vk`
491	`and Vi, Vj, Vk`
495	`or Vi, Vj, Vk`
496	`or Vi, Vj, Vk`

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
500	`xor Vi, Vj, Vk`
501	`xor Vi, Vj, Vk`
505	`not Vj, Vk`
506	`not Vj, Vk`
510	`neg.b Vj, Vk`
511	`neg.b Vj, Vk`
515	`neg.h Vj, Vk`
516	`neg.h Vj, Vk`
520	`neg.w Vj, Vk`
521	`neg.w Vj, Vk`
525	`neg.l Vj, Vk`
526	`neg.l Vj, Vk`
530	`neg.s Vj, Vk` (native mode)
531	`neg.s Vj, Vk` (native mode)
535	`neg.d Vj, Vk` (native mode)
536	`neg.d Vj, Vk` (native mode)
770	`merg.t Vi, Sj, Vk`
771	`merg.t Vi, Sj, Vk`
775	`merg.f Vi, Sj, Vk`
776	`merg.f Vi, Sj, Vk`
780	`mask.t Vi, Sj, Vk`
781	`mask.t Vi, Sj, Vk`
785	`mask.f Vi, Sj, Vk`
786	`mask.f Vi, S,, Vk`
790	`merg.t Vi, Vj, Vk`
791	`merg.t Vi, Vj, Vk`
800	`mask.t Vi, Vj, Vk`
801	`mask.t Vi, Vj, Vk`

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
805	`plc.t Vj, Vk`
806	`plc.t Vj, Vk`
810	`cprs.t Vi, Vj, Vk`
811	`cprs.t Vi, Vj, Vk`
815	`cprs.t Vi, Vj, Vk`
816	`cprs.t Vi, Vj, Vk`
820	`sum.b Vk`
821	`sum.b Vk`
825	`sum.h Vk`
826	`sum.h Vk`
830	`sum.w Vk`
831	`sum.w Vk`
835	`sum.l Vk`
836	`sum.l Vk`
840	`sum.s Vk` (native mode)
841	`sum.s Vk` (native mode)
845	`sum.d Vk` (native mode)
846	`sum.d Vk` (native mode)
880	`max.b Vk`
881	`max.b Vk`
885	`max.h Vk`
886	`max.h Vk`
890	`max.w Vk`
891	`max.w Vk`
895	`max.l Vk`
896	`max.l Vk`
900	`max.s Vk` (native mode)
901	`max.s Vk` (native mode)

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
905	`max.d Vk` (native mode)
906	`max.d Vk` (native mode)
910	`min.b Vk`
911	`min.b Vk`
915	`min.h Vk`
916	`min.h Vk`
920	`min.w Vk`
921	`min.w Vk`
925	`min.l Vk`
926	`min.l Vk`
930	`min.s Vk` (native mode)
931	`min.s Vk` (native mode)
935	`min.d Vk` (native mode)
936	`min.d Vk` (native mode)
940	`all Vk`
941	`all Vk`
950	`any Vk`
951	`any Vk`
960	`parity Vk`
961	`parity Vk`
1125	`le.s Sj, Vk` (IEEE mode)
1126	`le.s Sj, Vk` (IEEE mode)
1130	`le.d Sj, Vk` (IEEE mode)
1131	`le.d Sj, Vk` (IEEE mode)
1155	`lt.s Sj, Vk` (IEEE mode)
1156	`lt.s Sj, Vk` (IEEE mode)
1160	`lt.d Sj, Vk` (IEEE mode)
1161	`lt.d Sj, Vk` (IEEE mode)

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
1185	`eq.s Sj, Vk` (IEEE mode)
1186	`eq.s Sj, Vk` (IEEE mode)
1190	`eq.d Sj, Vk` (IEEE mode)
1191	`eq.d Sj, Vk` (IEEE mode)
1270	`add.s Vi, Sj, Vk` (IEEE mode)
1271	`add.s Vi, Sj, Vk` (IEEE mode)
1275	`add.d Vi, Sj, Vk` (IEEE mode)
1276	`add.d Vi, Sj, Vk` (IEEE mode)
1300	`sub.s Vi, Sj, Vk` (IEEE mode)
1301	`sub.s Vi, Sj, Vk` (IEEE mode)
1305	`sub.d Vi, Sj, Vk` (IEEE mode)
1306	`sub.d Vi, Sj, Vk` (IEEE mode)
1350	`le.s Vj, Vk` (IEEE mode)
1351	`le.s Vj, Vk` (IEEE mode)
1355	`le.d Vj, Vk` (IEEE mode)
1356	`le.d Vj, Vk` (IEEE mode)
1380	`lt.s Vj, Vk` (IEEE mode)
1381	`lt.s Vj, Vk` (IEEE mode)
1385	`lt.d Vj, Vk` (IEEE mode)
1386	`lt.d Vj, Vk` (IEEE mode)
1410	`eq.s Vj, Vk` (IEEE mode)
1411	`eq.s Vj, Vk` (IEEE mode)
1415	`eq.d Vj, Vk` (IEEE mode)
1416	`eq.d Vj, Vk` (IEEE mode)
1450	`add.s Vi, Vj, Vk` (IEEE mode)
1451	`add.s Vi, Vj, Vk` (IEEE mode)
1455	`add.d Vi, Vj, Vk` (IEEE mode)
1456	`add.d Vi, Vj, Vk` (IEEE mode)

Table 103 (continued)
cpu4041 class 2 subtests

Subtest	Instruction tested/test performed
1480	`sub.s Vi, Vj, Vk` (IEEE mode)
1481	`sub.s Vi, Vj, Vk` (IEEE mode)
1485	`sub.d Vi, Vj, Vk` (IEEE mode)
1486	`sub.d Vi, Vj, Vk` (IEEE mode)
1530	`neg.s Vj, Vk` (IEEE mode)
1531	`neg.s Vj, Vk` (IEEE mode)
1535	`neg.d Vj, Vk` (IEEE mode)
1536	`neg.d Vj, Vk` (IEEE mode)
1840	`sum.s Vk` (IEEE mode)
1841	`sum.s Vk` (IEEE mode)
1845	`sum.d Vk` (IEEE mode)
1846	`sum.d Vk` (IEEE mode)
1900	`max.s Vk` (IEEE mode)
1901	`max.s Vk` (IEEE mode)
1905	`max.d Vk` (IEEE mode)
1906	`max.d Vk` (IEEE mode)
1930	`min.s Vk` (IEEE mode)
1931	`min.s Vk` (IEEE mode)
1935	`min.d Vk` (IEEE mode)
1936	`min.d Vk` (IEEE mode)

Class 3 subtests

Class 3 subtests, listed in Table 104, verify the operation of the multiply and divide pipe. Specifically, this class verifies the vector/vector and scalar/vector multiplications and divisions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 104
cpu4041 class 3 subtests

Subtest	Instruction tested/test performed
630	`mul.b Vi, Sj, Vk`
631	`mul.b Vi, Sj, Vk`
635	`mul.h Vi, S, Vk`
636	`mul.h Vi, S, Vk`
640	`mul.w Vi, Sj, Vk`
641	`mul.w Vi, Sj, Vk`
645	`mul.l Vi, Sj, Vk`
646	`mul.l Vi, Sj, Vk`
650	`mul.s Vi, Sj, Vk` (native mode)
651	`mul.s Vi, Sj, Vk` (native mode)
655	`mul.d Vi, Sj, Vk` (native mode)
656	`mul.d Vi, Sj, Vk` (native mode)
660	`div.b Vi, Sj, Vk`
661	`div.b Vi, Sj, Vk`
665	`div.h Vi, Sj, Vk`
666	`div.h Vi, Sj, Vk`
670	`div.w Vi, Sj, Vk`
671	`div.w Vi, Sj, Vk`
675	`div.l Vi, Sj, Vk`
676	`div.l Vi, Sj, Vk`

Table 104 (continued)
cpu4041 class 3 subtests

Subtest	Instruction tested/test performed
680	`div.s Vi, Sj, Vk` (native mode)
681	`div.s Vi, Sj, Vk` (native mode)
685	`div.d Vi, Sj, Vk` (native mode)
686	`div.d Vi, Sj, Vk` (native mode)
700	`mul.b Vi, Vj, Vk`
701	`mul.b Vi, Vj, Vk`
705	`mul.h Vi, Vj, Vk`
706	`mul.h Vi, Vj, Vk`
710	`mul.w Vi, Vj, Vk`
711	`mul.w Vi, Vj, Vk`
715	`mul.l Vi, Vj, Vk`
716	`mul.l Vi, Vj, Vk`
720	`mul.s Vi, Vj, Vk` (native mode)
721	`mul.s Vi, Vj, Vk` (native mode)
725	`mul.d Vi, Vj, Vk` (native mode)
726	`mul.d Vi, Vj, Vk` (native mode)
740	`div.b Vi, Vj, Vk`
741	`div.b Vi, Vj, Vk`
745	`div.h Vi, Vj, Vk`
746	`div.h Vi, Vj, Vk`
750	`div.w Vi, Vj, Vk`
751	`div.w Vi, Vj, Vk`
755	`div.l Vi, Vj, Vk`
756	`div.l Vi, Vj, Vk`
760	`div.s Vi, Vj, Vk` (native mode)
761	`div.s Vi, Vj, Vk` (native mode)
765	`div.d Vi, Vj, Vk` (native mode)
766	`div.d Vi, Vj, Vk` (native mode)

Table 104 (continued)
cpu4041 class 3 subtests

Subtest	Instruction tested/test performed
850	`prod.b Vk`
851	`prod.b Vk`
855	`prod.h Vk`
856	`prod.h Vk`
860	`prod.w Vk`
861	`prod.w Vk`
865	`prod.l Vk`
866	`prod.l Vk`
870	`prod.s Vk` (native mode)
871	`prod.s Vk` (native mode)
875	`prod.d Vk` (native mode)
876	`prod.d Vk` (native mode)
1650	`mul.s Vi, Sj, Vk` (IEEE mode)
1651	`mul.s Vi, Sj, Vk` (IEEE mode)
1655	`mul.d Vi, Sj, Vk` (IEEE mode)
1656	`mul.d Vi, Sj, Vk` (IEEE mode)
1680	`div.s Vi, Sj, Vk` (IEEE mode)
1681	`div.s Vi, Sj, Vk` (IEEE mode)
1685	`div.d Vi, Sj, Vk` (IEEE mode)
1686	`div.d Vi, Sj, Vk` (IEEE mode)
1720	`mul.s Vi, Vj, Vk` (IEEE mode)
1721	`mul.s Vi, Vj, Vk` (IEEE mode)
1725	`mul.d Vi, Vj, Vk` (IEEE mode)
1726	`mul.d Vi, Vj, Vk` (IEEE mode)
1760	`div.s Vi, Vj, Vk` (IEEE mode)
1761	`div.s Vi, Vj, Vk` (IEEE mode)
1765	`div.d Vi, Vj, Vk` (IEEE mode)
1766	`div.d Vi, Vj, Vk` (IEEE mode)

Table 104 (continued)
cpu4041 class 3 subtests

Subtest	Instruction tested/test performed
1870	`prod.s Vk` (IEEE mode)
1871	`prod.s Vk` (IEEE mode)
1875	`prod.d Vk` (IEEE mode)
1876	`prod.d Vk` (IEEE mode)

Class 4 subtests

Class 4 subtests, listed in Table 105, verify the operation of loading and storing vector registers. Specifically, this class verifies loading and storing the vector using direct addressing and vector of indices, and storing of vectors and scalar registers using the extended operations.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 105
cpu4041 class 4 subtests

Subtest	Instruction tested/test performed
200	`ld.b <effa>, Vk`
201	`ld.b <effa>, Vk`
205	`ld.h <effa>, Vk`
206	`ld.h <effa>, Vk`
210	`ld.w <effa>, Vk`
211	`ld.w <effa>, Vk`
215	`ld.l <effa>, Vk`
216	`ld.l <effa>, Vk`
230	`st.b Vk, <effa>`
231	`st.b Vk, <effa>`
235	`st.h Vk, <effa>`
236	`st.h Vk, <effa>`
240	`st.w Vk, <effa>`
241	`st.w Vk, <effa>`
245	`st.l Vk, <effa>`
246	`st.l Vk, <effa>`
550	`ldvi.b Vj, Vk`
551	`ldvi.b Vj, Vk`
555	`ldvi.h Vj, Vk`
556	`ldvi.h Vj, Vk`

Table 105 (continued)
cpu4041 class 4 subtests

Subtest	Instruction tested/test performed
560	`ldvi.w Vj, Vk`
561	`ldvi.w Vj, Vk`
565	`ldvi.l Vj, Vk`
566	`ldvi.l Vj, Vk`
570	`stvi.b Vi, Vj`
571	`stvi.b Vi, Vj`
575	`stvi.b Vi, Vj`
576	`stvi.b Vi, Vj`
580	`stvi.w Vi, Vj`
581	`stvi.w Vi, Vj`
585	`stvi.l Vi, Vj`
586	`stvi.l Vi, Vj`
590	`stvi.b Si, Vj`
591	`stvi.b Si, Vj`
595	`stvi.h Si, Vj`
596	`stvi.h Si, Vj`
600	`stvi.w Si, Vj`
601	`stvi.w Si, Vj`
605	`stvi.l Si, Vj`
606	`stvi.l Si, Vj`
610	`ste.b Sk, <effa>`
611	`ste.b Sk, <effa>`
615	`ste.h Sk, <effa>`
616	`ste.h Sk, <effa>`
620	`ste.w Vk, <effa>`
621	`ste.w Vk, <effa>`
625	`ste.l Vk, <effa>`
626	`ste.l Vk, <effa>`

Enhanced vector instruction tests (cpu4241)

13

The cpu4241 test is a group of vector instruction tests used to verify the operation of vector and vector-under-mask instructions, along with their interfaces to other C3400 Series subsystems. The verification is performed by exercising each vector/vector and scalar/vector instruction while varying all of the parameters on which the instructions depend.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 106.

Table 106
`cpu4241` functional areas tested

Functional area	Tested by diagnostic	Exercised by the diagnostic
CPU	Yes	No
CUJ	Yes	No
Memory even	No	Yes
Memory odd	No	Yes
PI2	No	Yes
CCU	No	No
SP5	No	Yes

In order to run the `cpu4241` test, the boards listed in Table 107 must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table 107
`cpu4241` required functional boards

Board	Test to verify
Service processor (SP5)	`spu1000, spu4000`
PBUS interface adapter (PI2)	`pi2_4000`
Memory system (MEM, MEM2, MEM3)	`mem4100`
Central processor unit (CPU)	`cpu4030`
CPU utility board (CUJ)	`CUJ_SCAN`

Note

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the cpu4241 test, use the procedure shown in Figure 167.

Figure 167
cpu4241 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL
: test cpu4241 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if `mem4100` was executed with failures, or if `spu4000` was executed, then `initall` must be executed prior to test execution, but only after the SPU EPROM based self-test has passed. Failure to execute `initall` in these circumstances could result in invalid test results.

Entering only

```
test cpu4241
```

executes all cpu4241 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the -c or -s options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 168.

Figure 168
`cpu4241` test parameter menu

```
Test 'cpu4241.t'                              Thu Nov 19 00:00:00 1985 2

                    ENTER TEST PARAMETERS
       [ ]    Encloses allowed input ranges or values
       ( )    Encloses the default value
       ^      Returns to the previous prompt
       :nn    Returns to the prompt # nn
       :      Returns to the first unsatisfied prompt
       :?     Reviews previous entries

1: Run default switches? [y,n]                                  (y) ->
2: CPUs to test: [01234567]                            (01234567) ->
3: Parallel Test Execution? [y,n]                               (y) ->
4: Forced Faulting Enabled? [y,n]                               (n) ->
5: Fault on Instruction Fetches? [y,n.                          (n) ->
6: Sequential Execution? [y,n]                                  (n) ->
7: Timeout Scale Factor Enabled? [1-100]                        (1) ->
8: Dcache Enabled? [y,n]                                        (y) ->
9: Segment of Execution? [0-7]                                  (0) ->
10: Chained Execution Mode? [y,n]                               (n) ->
11: Loop Enabled? [y,n]                                         (n) ->
12: Hard Errors Enabled? [y,n]                                  (y) ->
13: Number of vl values to test(0..vl)? [0-128]               (128) ->
14: Load CPU Code? [y,n]                                        (y) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

In Figure 168, prompt 2, `[01234567]` represents all available CPUs in the machine under test.

Prompt explanations

A description of the meaning of each prompt follows:

```
1: Run default switches? [y/n]                (y) ->
```

If you respond with **y** or **RETURN**, no additional test parameter prompts are displayed and testing begins.

However, if you respond with **n**, additional test parameter prompts are displayed allowing choices other than the default selections. The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
2: CPUs to test: [01234567]                (01234567)->
```

This prompt allows selection of the CPUs to be used in the test. The possible selections, and the default, represented by `01234567`, consists of all available CPUs.

```
3: Parallel Test Execution? [y,n]             (y) ->
```

If the system has more than one CPU available for testing, then this prompt appears. If answered with **y**, then each CPU selected for testing executes the tests concurrently; otherwise, each subtest is executed on the selected CPUs in the order the CPUs were selected.

```
4: Forced Faulting Enabled? [y,n]             (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system forces a nonresident data exception to occur on every data reference.

If this option is enabled, subtest execution time is increased and the timeout scale factor requires adjustment to prevent the SPU from terminating the test prematurely.

```
5: Fault on Instruction Fetches? [y,n]        (n) ->
```

In answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is only supplied if the previous prompt is answered with **y**.

```
6: Sequential Execution? [y,n]                (n) ->
```

If set by entering **y**, the sequential bit in the processor status word (PSW) is set to forced sequential execution mode.

```
7: Timeout Scale Factor [1-100]                (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number entered. For example, if **5** is entered, the normal timeout factor is multiplied by 5, and it takes the test five times as long to timeout.

```
8: Dcache Enabled? [y,n]                       (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

```
9: Segment of Execution? [0-7]                 (0) ->
```

This prompt is only displayed if forced faulting is not enabled. The segment of execution is contained in bits <31..29> of the program counter (PC).

- If **0** is entered, then bits <31..29> of the PC are `000` and the test is run in ring zero.
- If **1** is entered, then bits <31..29> of the PC are `001` and the test is run in ring one.
- If **2** is entered, then bits <31..29> of the PC are `010` and the test is run in ring two.
- If **3** is entered, then bits <31..29> of the PC are `011` and the test is run in ring three.
- If **4**, **5**, **6**, or **7** is entered, then bits <31..29> of the PC are `100`, `101`, `110`, and `111` respectively and the test is run in ring four.

Refer to the *CONVEX Architecture Reference Manual (C Series)* for more information concerning the meaning of rings in the machine architecture.

```
10: Chained Execution Mode? [y,n]              (n) ->
```

With this option enabled, the test is executed in chained mode, which causes the CPU to perform subtest sequencing. The service processor is unaware of the action of the CPU, unless a subtest fails or all of the subtests pass. If this option is enabled, test execution time is greatly reduced.

The only information printed to the console upon completion is a pass or fail message. Also, this option cannot be enabled if the `-c` or the `-s` options were used in the invocation procedure. This option does not disable any subtests from begin executed.

```
11: Loop Enabled? [y,n]                    (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by pressing **CTRL- C**.

```
12: Hard Errors Enabled? [y,n]             (y) ->
```

If this option is enabled and a hard error occurs, the clocks are stopped and the test fails. If this option is disabled, parity errors and other sources of hard errors go undetected. It is recommended that hard errors normally be enabled.

```
13: Number of vl values to test(0..vl)?[0-128] (128)->
```

Each vector instruction which is dependent on the vector length (VL), has two separate subtests. The first subtest executes with a hard coded VL of 128. The second subtest uses this user-supplied value as the initial VL value. The subtest is then executed and the VL decremented. This is repeated until the VL under test becomes less than zero.

```
14: Load CPU Code? [y,n]                   (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code is not reloaded, thus saving time.

Test parameter summary

When all prompts have been answered, the screen displays a test parameter summary which echoes the prompts that have been answered, as illustrated in Figure 169.

Figure 169
`cpu4241` test parameter summary

```
                TEST PARAMETER SUMMARY

Run default switches?                          : n
Cpus to test:                                  : 01
Parallel Test Execution?                       : y
Forced Faulting Enabled?                       : y
Sequential Execution?                          : n
Timeout Scale Factor Enabled?                  : 1
Dcache Enabled?                                : y
Segment of Execution?                          : 0
Chained Execution Mode?                        : n
Loop Enabled?                                  : n
Hard Errors Enabled?                           : y
Number of vl values to test (0..vl)?           : 128
Load CPU Code?                                 : y
```

The actual summary varies depending on the answers to the prompts.

Hardware initialization sequence

After the last prompt is entered by the user (and before test code execution) the following events occur:

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CUJ, PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- Clocks are turned on for the processor selected. If parallel execution is selected, each CPU's clocks are turned on at one time. If sequential execution is selected, each CPU's clocks are turned on one at a time.

Note

The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

Current memory allocation

Immediately before test code execution, a current memory allocation screen is displayed. Figure 170 is an example of the current memory allocation screen.

Figure 170
`cpu4241` current memory allocation screen

```
                    Current Memory Allocation

File No. Physical Address Pid File Name                Logical Offset
      1  00000000-00027fff  0 p0r0_4241               00000000
      2  00028000-000b1fff  0 cpu4241.rnn             00022000
      1  000b2000-000d9fff  1 p0r0_4241               00000000
      2  000da000-00163fff  1 cpu4241.rnn             00022000
         03ff7000-03ff9fff  1 ptet   NA
         03ffa000-03ffafff  1 pte2   NA
         03ffb000-03ffdfff  0 ptet   NA
         03ffe000-03ffefff  0 pte2   NA
         03fffc00-3fffffff  1 pte1   NA
```

The physical and logical addresses shown, as well as the file names, are only representative. The actual addresses vary depending on installed memory and file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First column**—File number.
- **Second column**—Physical memory addresses where the specified file is loaded (useful in conjunction with the mm(1d) utility).
- **Third column**—Process identification.
- **Fourth column**—File name. (The actual path is */filename* or /mnt/test/CPU/*filename*. The entries `pte1`, `pte2`, and `ptet` are not actual files, but are indications of the page tables.)
- **Fifth column**—Logical starting address of the specified file.

Subtest descriptions

There are four classes of subtests for `cpu4241`.

In the tables in this section, the "Instructions tested/test performed" column lists each instruction that is tested. For more information on individual instructions, refer to the *CONVEX Architecture Reference Manual (C Series).* The object module (the executable code) for all classes is cpu4241.rnn.

Class 1 subtests

Class 1 subtests, listed in Table 108, verify the operation of loading, storing, and modifying the vector unit control functions. Specifically, this class verifies the ability to alter and save the vector length (VL) register, vector stride (VS) register, and the vector merge (VM) register.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 108
`cpu4241` class 1 subtests

Subtest	Instruction tested/test performed
21	`mov.w VL, Sk`
26	`mov.w VS, Sk`
61	`mov Sk, VM(U|L)`
62	`mov VM(U|L), Sk`
1005	Vector valid test

Class 2 subtests

Class 2 subtests, listed in Table 109, verify the operation of the logical and arithmetic pipes. Specifically, this class verifies vector/vector and scalar/vector comparisons, vector/vector and scalar/vector additions and subtractions, and vector reductions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 109
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
2105	`le.b.t Sj, Vk`
2106	`le.b.t Sj, Vk`
4105	`le.b.f Sj, Vk`
4106	`le.b.f Sj, Vk`
2110	`le.h.t Sj, Vk`
2111	`le.h.t Sj, Vk`
4110	`le.h.f Sj, Vk`
4111	`le.h.f Sj, Vk`
2115	`le.w.t Sj, Vk`
2116	`le.w.t Sj, Vk`
4115	`le.w.f Sj, Vk`
4116	`le.w.f Sj, Vk`
2120	`le.l.t Sj, Vk`
2121	`le.l.t Sj, Vk`
4120	`le.l.f Sj, Vk`
4121	`le.l.f Sj, Vk`
2125	`le.s.t Sj, Vk` (native mode)
2126	`le.s.t Sj, Vk` (native mode)
3125	`le.s.t Sj, Vk` (IEEE mode)
3126	`le.s.t Sj, Vk` (IEEE mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
4125	`le.s.f Sj, Vk` (native mode)
4126	`le.s.f Sj, Vk` (native mode)
5125	`le.s.f Sj, Vk` (IEEE mode)
5126	`le.s.f Sj, Vk` (IEEE mode)
2130	`le.d.t Sj, Vk` (native mode)
2131	`le.d.t Sj, Vk` (native mode)
3130	`le.d.t Sj, Vk` (IEEE mode)
3131	`le.d.t Sj, Vk` (IEEE mode)
4130	`le.d.f Sj, Vk` (native mode)
4131	`le.d.f Sj, Vk` (native mode)
5130	`le.d.f Sj, Vk` (IEEE mode)
5131	`le.d.f Sj, Vk` (IEEE mode)
2135	`lt.b.t Sj, Vk`
2136	`lt.b.t Sj, Vk`
4135	`lt.b.f Sj, Vk`
4136	`lt.b.f Sj, Vk`
2140	`lt.h.t Sj, Vk`
2141	`lt.h.t Sj, Vk`
4140	`lt.h.f Sj, Vk`
4141	`lt.h.f Sj, Vk`
2145	`lt.w.t Sj, Vk`
2146	`lt.w.t Sj, Vk`
4145	`lt.w.f Sj, Vk`
4146	`lt.w.f Sj, Vk`
2150	`lt.l.t Sj, Vk`
2151	`lt.l.t Sj, Vk`
4150	`lt.l.f Sj, Vk`
4151	`lt.l.f Sj, Vk`

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
2155	`lt.s.t Sj, Vk` (native mode)
2156	`lt.s.t Sj, Vk` (native mode)
3155	`lt.s.t Sj, Vk` (IEEE mode)
3156	`lt.s.t Sj, Vk` (IEEE mode)
4155	`lt.s.f Sj, Vk` (native mode)
4156	`lt.s.f Sj, Vk` (native mode)
5155	`lt.s.f Sj, Vk` (IEEE mode)
5156	`lt.s.f Sj, Vk` (IEEE mode)
2160	`lt.d.t Sj, Vk` (native mode)
2161	`lt.d.t Sj, Vk` (native mode)
3160	`lt.d.t Sj, Vk` (IEEE mode)
3161	`lt.d.t Sj, Vk` (IEEE mode)
4160	`lt.d.f Sj, Vk` (native mode)
4161	`lt.d.f Sj, Vk` (native mode)
5160	`lt.d.f Sj, Vk` (IEEE mode)
5161	`lt.d.f Sj, Vk` (IEEE mode)
2165	`eq.b.t Sj, Vk`
2166	`eq.b.t Sj, Vk`
4165	`eq.b.f Sj, Vk`
4166	`eq.b.f Sj, Vk`
2170	`eq.h.t Sj, Vk`
2171	`eq.h.t Sj, Vk`
4170	`eq.h.f Sj, Vk`
4171	`eq.h.f Sj, Vk`
2175	`eq.w.t Sj, Vk`
2176	`eq.w.t Sj, Vk`
4175	`eq.w.f Sj, Vk`
4176	`eq.w.f Sj, Vk`

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
4180	`eq.l.f Sj, Vk`
4181	`eq.l.f Sj, Vk`
2185	`eq.s.t Sj, Vk` (native mode)
2186	`eq.s.t Sj, Vk` (native mode)
3185	`eq.s.t Sj, Vk` (IEEE mode)
3186	`eq.s.t Sj, Vk` (IEEE mode)
4185	`eq.s.f Sj, Vk` (native mode)
4186	`eq.s.f Sj, Vk` (native mode)
5185	`eq.s.f Sj, Vk` (IEEE mode)
5186	`eq.s.f Sj, Vk` (IEEE mode)
2190	`eq.d.t Sj, Vk` (native mode)
2191	`eq.d.t Sj, Vk` (native mode)
3190	`eq.d.t Sj, Vk` (IEEE mode)
3191	`eq.d.t Sj, Vk` (IEEE mode)
4190	`eq.d.f Sj, Vk` (native mode)
4191	`eq.d.f Sj, Vk` (native mode)
5190	`eq.d.f Sj, Vk` (IEEE mode)
5191	`eq.d.f Sj, Vk` (IEEE mode)
196	`shf Vi, Sj, Vk`
197	`shf Vi, Sj, Vk`
2196	`shf.t Vi, Sj, Vk`
2197	`shf.t Vi, Sj, Vk`
4196	`shf.f Vi, Sj, Vk`
4197	`shf.f Vi, Sj, Vk`
220	`tzc Vj, Vk`
221	`tzc Vj, Vk`
2220	`tzc.t Vj, Vk`
2221	`tzc.t Vj, Vk`

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
4220	tzc.f Vj, Vk
4221	tzc.f Vj, Vk
225	tzc Vj, Vk
226	tzc Vj, Vk
2225	tzc.t Vj, Vk
2226	tzc.t Vj, Vk
4225	tzc.f Vj, Vk
4226	tzc.f Vj, Vk
2250	add.b.t Vi, Sj, Vk
2251	add.b.t Vi, Sj, Vk
4250	add.b.f Vi, Sj, Vk
4251	add.b.f Vi, Sj, Vk
2255	add.h.t Vi, Sj, Vk
2256	add.h.t Vi, Sj, Vk
4255	add.h.f Vi, Sj, Vk
4256	add.h.f Vi, Sj, Vk
2260	add.w.t Vi, Sj, Vk
2261	add.w.t Vi, Sj, Vk
4260	add.w.f Vi, Sj, Vk
4261	add.w.f Vi, Sj, Vk
2265	add.l.t Vi, Sj, Vk
2266	add.l.t Vi, Sj, Vk
4265	add.l.f Vi, Sj, Vk
4266	add.l.f Vi, Sj, Vk
2270	add.s.t Vi, Sj, Vk (native mode)
2271	add.s.t Vi, Sj, Vk (native mode)
3270	add.s.t Vi, Sj, Vk (IEEE mode)
3271	add.s.t Vi, Sj, Vk (IEEE mode)

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
4270	add.s.f Vi, Sj, Vk (native mode)
4271	add.s.f Vi, Sj, Vk (native mode)
5270	add.s.f Vi, Sj, Vk (IEEE mode)
5271	add.s.f Vi, Sj, Vk (IEEE mode)
2275	add.d.t Vi, Sj, Vk (native mode)
2276	add.d.t Vi, Sj, Vk (native mode)
3275	add.d.t Vi, Sj, Vk (IEEE mode)
3276	add.d.t Vi, Sj, Vk (IEEE mode)
4275	add.d.f Vi, Sj, Vk (native mode)
4276	add.d.f Vi, Sj, Vk (native mode)
5275	add.d.f Vi, Sj, Vk (IEEE mode)
5276	add.d.f Vi, Sj, Vk (IEEE mode)
2280	sub.b.t Vi, Sj, Vk
2281	sub.b.t Vi, Sj, Vk
4280	sub.b.f Vi, Sj, Vk
4281	sub.b.f Vi, Sj, Vk
2285	sub.h.t Vi, Sj, Vk
2286	sub.h.t Vi, Sj, Vk
4285	sub.h.f Vi, Sj, Vk
4286	sub.h.f Vi, Sj, Vk
2290	sub.w.t Vi, Sj, Vk
2291	sub.w.t Vi, Sj, Vk
4290	sub.w.f Vi, Sj, Vk
4291	sub.w.f Vi, Sj, Vk
2295	sub.l.t Vi, Sj, Vk
2296	sub.l.t Vi, Sj, Vk
4295	sub.l.f Vi, Sj, Vk
4296	sub.l.f Vi, Sj, Vk

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
2300	sub.s.t Vi, Sj, Vk (native mode)
2301	sub.s.t Vi, Sj, Vk (native mode)
3300	sub.s.t Vi, Sj, Vk (IEEE mode)
3301	sub.s.t Vi, Sj, Vk (IEEE mode)
4300	sub.s.f Vi, Sj, Vk (native mode)
4301	sub.s.f Vi, Sj, Vk (native mode)
5300	sub.s.f Vi, Sj, Vk (IEEE mode)
5301	sub.s.f Vi, Sj, Vk (IEEE mode)
2305	sub.d.t Vi, Sj, Vk (native mode)
2306	sub.d.t Vi, Sj, Vk (native mode)
3305	sub.d.t Vi, Sj, Vk (IEEE mode)
3306	sub.d.t Vi, Sj, Vk (IEEE mode)
4305	sub.d.f Vi, Sj, Vk (native mode)
4306	sub.d.f Vi, Sj, Vk (native mode)
5305	sub.d.f Vi, Sj, Vk (IEEE mode)
5306	sub.d.f Vi, Sj, Vk (IEEE mode)
2310	and.t Vi, Sj, Vk
2311	and.t Vi, Sj, Vk
4310	and.f Vi, Sj, Vk
4311	and.f Vi, Sj, Vk
2315	or.t Vi, Sj, Vk
2316	or.t Vi, Sj, Vk
4315	or.f Vi, Sj, Vk
4316	or.f Vi, Sj, Vk
2320	xor.t Vi, Sj, Vk
2321	xor.t Vi, Sj, Vk
4320	xor.f Vi, Sj, Vk
4321	xor.f Vi, Sj, Vk

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
2325	`shf.t Sj, Vk`
2326	`shf.t Sj, Vk`
4325	`shf.f Sj, Vk`
4326	`shf.f Sj, Vk`
2330	`le.b.t Vj, Vk`
2331	`le.b.t Vj, Vk`
4330	`le.b.f Vj, Vk`
4331	`le.b.f Vj, Vk`
2335	`le.h.t Vj, Vk`
2336	`le.h.t Vj, Vk`
4335	`le.h.f Vj, Vk`
4336	`le.h.f Vj, Vk`
2340	`le.w.t Vj, Vk`
2341	`le.w.t Vj, Vk`
4340	`le.w.f Vj, Vk`
4341	`le.w.f Vj, Vk`
2345	`le.l.t Vj, Vk`
2346	`le.l.t Vj, Vk`
4345	`le.l.f Vj, Vk`
4346	`le.l.f Vj, Vk`
2350	`le.s.t Vj, Vk` (native mode)
2351	`le.s.t Vj, Vk` (native mode)
3350	`le.s.t Vj, Vk` (IEEE mode)
3351	`le.s.t Vj, Vk` (IEEE mode)
4350	`le.s.f Vj, Vk` (native mode)
4351	`le.s.f Vj, Vk` (native mode)
5350	`le.s.f Vj, Vk` (IEEE mode)
5351	`le.s.f Vj, Vk` (IEEE mode)

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
2355	`le.d.t Vj, Vk` (native mode)
2356	`le.d.t Vj, Vk` (native mode)
3355	`le.d.t Vj, Vk` (IEEE mode)
3356	`le.d.t Vj, Vk` (IEEE mode)
4355	`le.d.f Vj, Vk` (native mode)
4356	`le.d.f Vj, Vk` (native mode)
5355	`le.d.f Vj, Vk` (IEEE mode)
5356	`le.d.f Vj, Vk` (IEEE mode)
2360	`lt.b.t Vj, Vk`
2361	`lt.b.t Vj, Vk`
4360	`lt.b.f Vj, Vk`
4361	`lt.b.f Vj, Vk`
2365	`lt.h.t Vj, Vk`
2366	`lt.h.t Vj, Vk`
4365	`lt.h.f Vj, Vk`
4366	`lt.h.f Vj, Vk`
2370	`lt.w.t Vj, Vk`
2371	`lt.w.t Vj, Vk`
4370	`lt.w.f Vj, Vk`
4371	`lt.w.f Vj, Vk`
2375	`lt.l.t Vj, Vk`
2376	`lt.l.t Vj, Vk`
4375	`lt.l.f Vj, Vk`
4376	`lt.l.f Vj, Vk`
2380	`lt.s.t Vj, Vk` (native mode)
2381	`lt.s.t Vj, Vk` (native mode)
3380	`lt.s.t Vj, Vk` (IEEE mode)
3381	`lt.s.t Vj, Vk` (IEEE mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
4380	`lt.s.f Vj, Vk` (native mode)
4381	`lt.s.f Vj, Vk` (native mode)
5380	`lt.s.f Vj, Vk` (IEEE mode)
5381	`lt.s.f Vj, Vk` (IEEE mode)
2385	`lt.d.t Vj, Vk` (native mode)
2386	`lt.d.t Vj, Vk` (native mode)
3385	`lt.d.t Vj, Vk` (IEEE mode)
3386	`lt.d.t Vj, Vk` (IEEE mode)
4385	`lt.d.f Vj, Vk` (native mode)
4386	`lt.d.f Vj, Vk` (native mode)
5385	`lt.d.f Vj, Vk` (IEEE mode)
5386	`lt.d.f Vj, Vk` (IEEE mode)
2390	`eq.b.t Vj, Vk`
2391	`eq.b.t Vj, Vk`
4390	`eq.b.f Vj, Vk`
4391	`eq.b.f Vj, Vk`
2395	`eq.h.t Vj, Vk`
2396	`eq.h.t Vj, Vk`
4395	`eq.h.f Vj, Vk`
4396	`eq.h.f Vj, Vk`
2400	`eq.w.t Vj, Vk`
2401	`eq.w.t Vj, Vk`
4400	`eq.w.f Vj, Vk`
4401	`eq.w.f Vj, Vk`
2405	`eq.l.t Vj, Vk`
2406	`eq.l.t Vj, Vk`
4405	`eq.l.f Vj, Vk`
4406	`eq.l.f Vj, Vk`

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
2410	`eq.s.t Vj, Vk` (native mode)
2411	`eq.s.t Vj, Vk` (native mode)
3410	`eq.s.t Vj, Vk` (IEEE mode)
3411	`eq.s.t Vj, Vk` (IEEE mode)
4410	`eq.s.f Vj, Vk` (native mode)
4411	`eq.s.f Vj, Vk` (native mode)
5410	`eq.s.f Vj, Vk` (IEEE mode)
5411	`eq.s.f Vj, Vk` (IEEE mode)
2415	`eq.d.t Vj, Vk` (native mode)
2416	`eq.d.t Vj, Vk` (native mode)
3415	`eq.d.t Vj, Vk` (IEEE mode)
3416	`eq.d.t Vj, Vk` (IEEE mode)
4415	`eq.d.f Vj, Vk` (native mode)
4416	`eq.d.f Vj, Vk` (native mode)
5415	`eq.d.f Vj, Vk` (IEEE mode)
5416	`eq.d.f Vj, Vk` (IEEE mode)
420	`frint.s Vj, Vk` (native mode)
421	`frint.s Vj, Vk` (native mode)
1420	`frint.s Vj, Vk` (IEEE mode)
1421	`frint.s Vj, Vk` (IEEE mode)
2420	`frint.s.t Vj, Vk` (native mode)
2421	`frint.s.t Vj, Vk` (native mode)
3420	`frint.s.t Vj, Vk` (IEEE mode)
3421	`frint.s.t Vj, Vk` (IEEE mode)
4420	`frint.s.f Vj, Vk` (native mode)
4421	`frint.s.f Vj, Vk` (native mode)
5420	`frint.s.f Vj, Vk` (IEEE mode)
5421	`frint.s.f Vj, Vk` (IEEE mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
425	`frint.d Vj, Vk` (native mode)
426	`frint.d Vj, Vk` (native mode)
1425	`frint.d Vj, Vk` (IEEE mode)
1426	`frint.d Vj, Vk` (IEEE mode)
2425	`frint.d.t Vj, Vk` (native mode)
2426	`frint.d.t Vj, Vk` (native mode)
3425	`frint.d.t Vj, Vk` (IEEE mode)
3426	`frint.d.t Vj, Vk` (IEEE mode)
4425	`frint.d.f Vj, Vk` (native mode)
4426	`frint.d.f Vj, Vk` (native mode)
5425	`frint.d.f Vj, Vk` (IEEE mode)
5426	`frint.d.f Vj, Vk` (IEEE mode)
2430	`add.b.t Vi, Vj, Vk`
2431	`add.b.t Vi, Vj, Vk`
4430	`add.b.f Vi, Vj, Vk`
4431	`add.b.f Vi, Vj, Vk`
2435	`add.h.t Vi, Vj, Vk`
2436	`add.h.t Vi, Vj, Vk`
4435	`add.h.f Vi, Vj, Vk`
4436	`add.h.f Vi, Vj, Vk`
2440	`add.w.t Vi, Vj, Vk`
2441	`add.w.t Vi, Vj, Vk`
4440	`add.w.f Vi, Vj, Vk`
4441	`add.w.f Vi, Vj, Vk`
2445	`add.l.t Vi, Vj, Vk`
2446	`add.l.t Vi, Vj, Vk`
4445	`add.l.f Vi, Vj, Vk`
4446	`add.l.f Vi, Vj, Vk`

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
2450	`add.s.t Vi, Vj, Vk` (native mode)
2451	`add.s.t Vi, Vj, Vk` (native mode)
3450	`add.s.t Vi, Vj, Vk` (IEEE mode)
3451	`add.s.t Vi, Vj, Vk` (IEEE mode)
4450	`add.s.f Vi, Vj, Vk` (native mode)
4451	`add.s.f Vi, Vj, Vk` (native mode)
5450	`add.s.f Vi, Vj, Vk` (IEEE mode)
5451	`add.s.f Vi, Vj, Vk` (IEEE mode)
2455	`add.d.t Vi, Vj, Vk` (native mode)
2456	`add.d.t Vi, Vj, Vk` (native mode)
3455	`add.d.t Vi, Vj, Vk` (IEEE mode)
3456	`add.d.t Vi, Vj, Vk` (IEEE mode)
4455	`add.d.f Vi, Vj, Vk` (native mode)
4456	`add.d.f Vi, Vj, Vk` (native mode)
5455	`add.d.f Vi, Vj, Vk` (IEEE mode)
5456	`add.d.f Vi, Vj, Vk` (IEEE mode)
2460	`sub.b.t Vi, Vj, Vk`
2461	`sub.b.t Vi, Vj, Vk`
4460	`sub.b.f Vi, Vj, Vk`
4461	`sub.b.f Vi, Vj, Vk`
2465	`sub.h.t Vi, Vj, Vk`
2466	`sub.h.t Vi, Vj, Vk`
4465	`sub.h.f Vi, Vj, Vk`
4466	`sub.h.f Vi, Vj, Vk`
2470	`sub.w.t Vi, Vj, Vk`
2471	`sub.w.t Vi, Vj, Vk`
4470	`sub.w.f Vi, Vj, Vk`
4471	`sub.w.f Vi, Vj, Vk`

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
2475	`sub.l.t Vi, Vj, Vk`
2476	`sub.l.t Vi, Vj, Vk`
4475	`sub.l.f Vi, Vj, Vk`
4476	`sub.l.f Vi, Vj, Vk`
2480	`sub.s.t Vi, Vj, Vk` (native mode)
2481	`sub.s.t Vi, Vj, Vk` (native mode)
3480	`sub.s.t Vi, Vj, Vk` (IEEE mode)
3481	`sub.s.t Vi, Vj, Vk` (IEEE mode)
4480	`sub.s.f Vi, Vj, Vk` (native mode)
4481	`sub.s.f Vi, Vj, Vk` (native mode)
5480	`sub.s.f Vi, Vj, Vk` (IEEE mode)
5481	`sub.s.f Vi, Vj, Vk` (IEEE mode)
2485	`sub.d.t Vi, Vj, Vk` (native mode)
2486	`sub.d.t Vi, Vj, Vk` (native mode)
3485	`sub.d.t Vi, Vj, Vk` (IEEE mode)
3486	`sub.d.t Vi, Vj, Vk` (IEEE mode)
4485	`sub.d.f Vi, Vj, Vk` (native mode)
4486	`sub.d.f Vi, Vj, Vk` (native mode)
5485	`sub.d.f Vi, Vj, Vk` (IEEE mode)
5486	`sub.d.f Vi, Vj, Vk` (IEEE mode)
2490	`and.t Vi, Vj, Vk`
2491	`and.t Vi, Vj, Vk`
4490	`and.f Vi, Vj, Vk`
4491	`and.f Vi, Vj, Vk`
2495	`or.t Vi, Vj, Vk`
2496	`or.t Vi, Vj, Vk`
4495	`or.f Vi, Vj, Vk`
4496	`or.f Vi, Vj, Vk`

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
2500	`xor.t Vi, Vj, Vk`
2501	`xor.t Vi, Vj, Vk`
4500	`xor.f Vi, Vj, Vk`
4501	`xor.f Vi, Vj, Vk`
2505	`not.t Vj, Vk`
2506	`not.t Vj, Vk`
4505	`not.f Vj, Vk`
4506	`not.f Vj, Vk`
2510	`neg.b.t Vj, Vk`
2511	`neg.b.t Vj, Vk`
4510	`neg.b.f Vj, Vk`
4511	`neg.b.f Vj, Vk`
2515	`neg.h.t Vj, Vk`
2516	`neg.h.t Vj, Vk`
4515	`neg.h.f Vj, Vk`
4516	`neg.h.f Vj, Vk`
2520	`neg.w.t Vj, Vk`
2521	`neg.w.t Vj, Vk`
4520	`neg.w.f Vj, Vk`
4521	`neg.w.f Vj, Vk`
2525	`neg.l.t Vj, Vk`
2526	`neg.l.t Vj, Vk`
4525	`neg.l.f Vj, Vk`
4526	`neg.l.f Vj, Vk`
2530	`neg.s.t Vj, Vk` (native mode)
2531	`neg.s.t Vj, Vk` (native mode)
3530	`neg.s.t Vj, Vk` (IEEE mode)
3531	`neg.s.t Vj, Vk` (IEEE mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
4530	`neg.s.f Vj, Vk` (native mode)
4531	`neg.s.f Vj, Vk` (native mode)
5530	`neg.s.f Vj, Vk` (IEEE mode)
5531	`neg.s.f Vj, Vk` (IEEE mode)
2535	`neg.d.t Vj, Vk` (native mode)
2536	`neg.d.t Vj, Vk` (native mode)
3535	`neg.d.t Vj, Vk` (IEEE mode)
3536	`neg.d.t Vj, Vk` (IEEE mode)
4535	`neg.d.f Vj, Vk` (native mode)
4536	`neg.d.f Vj, Vk` (native mode)
5535	`neg.d.f Vj, Vk` (IEEE mode)
5536	`neg.d.f Vj, Vk` (IEEE mode)
540	`sub.s Si, Vj, Vk` (native mode)
541	`sub.s Si, Vj, Vk` (native mode)
1540	`sub.s Si, Vj, Vk` (IEEE mode)
1541	`sub.s Si, Vj, Vk` (IEEE mode)
2540	`sub.s.t Si, Vj, Vk` (native mode)
2541	`sub.s.t Si, Vj, Vk` (native mode)
3540	`sub.s.t Si, Vj, Vk` (IEEE mode)
3541	`sub.s.t Si, Vj, Vk` (IEEE mode)
4540	`sub.s.f Si, Vj, Vk` (native mode)
4541	`sub.s.f Si, Vj, Vk` (native mode)
5540	`sub.s.f Si, Vj, Vk` (IEEE mode)
5541	`sub.s.f Si, Vj, Vk` (IEEE mode)
545	`sub.d Si, Vj, Vk` (native mode)
546	`sub.d Si, Vj, Vk` (native mode)
1545	`sub.d Si, Vj, Vk` (IEEE mode)
1546	`sub.d Si, Vj, Vk` (IEEE mode)

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
2545	sub.d.t Si, Vj, Vk (native mode)
2546	sub.d.t Si, Vj, Vk (native mode)
3545	sub.d.t Si, Vj, Vk (IEEE mode)
3546	sub.d.t Si, Vj, Vk (IEEE mode)
4545	sub.d.f Si, Vj, Vk (native mode)
4546	sub.d.f Si, Vj, Vk (native mode)
5545	sub.d.f Si, Vj, Vk (IEEE mode)
5546	sub.d.f Si, Vj, Vk (IEEE mode)
795	shf Vi, Vj, Vk
796	shf Vi, Vj, Vk
2795	shf.t Vi, Vj, Vk
2796	shf.t Vi, Vj, Vk
4795	shf.f Vi, Vj, Vk
4796	shf.f Vi, Vj, Vk
2805	plc.t.t Vj, Vk
2806	plc.t.t Vj, Vk
4805	plc.t.f Vj, Vk
4806	plc.t.f Vj, Vk
2815	xpnd.t.t Vj, Vk
2816	xpnd.t.t Vj, Vk
4815	xpnd.t.f Vj, Vk
4816	xpnd.t.f Vj, Vk
2820	sum.b.t Vk
2821	sum.b.t Vk
4820	sum.b.f Vk
4821	sum.b.f Vk
2825	sum.h.t Vk
2826	sum.h.t Vk

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
4825	`sum.h.f Vk`
4826	`sum.h.f Vk`
2830	`sum.w.t Vk`
2831	`sum.w.t Vk`
4830	`sum.w.f Vk`
4831	`sum.w.f Vk`
2835	`sum.l.t Vk`
2836	`sum.l.t Vk`
4835	`sum.l.f Vk`
4836	`sum.l.f Vk`
2840	`sum.s.t Vk` (native mode)
2841	`sum.s.t Vk` (native mode)
3840	`sum.s.t Vk` (IEEE mode)
3841	`sum.s.t Vk` (IEEE mode)
4840	`sum.s.f Vk` (native mode)
4841	`sum.s.f Vk` (native mode)
5840	`sum.s.f Vk` (IEEE mode)
5841	`sum.s.f Vk` (IEEE mode)
2845	`sum.d.t Vk` (native mode)
2846	`sum.d.t Vk` (native mode)
3845	`sum.d.t Vk` (IEEE mode)
3846	`sum.d.t Vk` (IEEE mode)
4845	`sum.d.f Vk` (native mode)
4846	`sum.d.f Vk` (native mode)
5845	`sum.d.f Vk` (IEEE mode)
5846	`sum.d.f Vk` (IEEE mode)
2880	`max.b.t Vk`
2881	`max.b.t Vk`

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
4880	max.b.f Vk
4881	max.b.f Vk
2885	max.h.t Vk
2886	max.h.t Vk
4885	max.h.f Vk
4886	max.h.f Vk
2890	max.w.t Vk
2891	max.w.t Vk
4890	max.w.f Vk
4891	max.w.f Vk
2895	max.l.t Vk
2896	max.l.t Vk
4895	max.l.f Vk
4896	max.l.f Vk
2900	max.s.t Vk (native mode)
2901	max.s.t Vk (native mode)
3900	max.s.t Vk (IEEE mode)
3901	max.s.t Vk (IEEE mode)
4900	max.s.f Vk (native mode)
4901	max.s.f Vk (native mode)
5900	max.s.f Vk (IEEE mode)
5901	max.s.f Vk (IEEE mode)
2905	max.d.t Vk (native mode)
2906	max.d.t Vk (native mode)
3905	max.d.t Vk (IEEE mode)
3906	max.d.t Vk (IEEE mode)
4905	max.d.f Vk (native mode)
4906	max.d.f Vk (native mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
5905	`max.d.f Vk` (IEEE mode)
5906	`max.d.f Vk` (IEEE mode)
2910	`min.b.t Vk`
2911	`min.b.t Vk`
4910	`min.b.f Vk`
4911	`min.b.f Vk`
2915	`min.h.t Vk`
2916	`min.h.t Vk`
4915	`min.h.f Vk`
4916	`min.h.f Vk`
2920	`min.w.t Vk`
2921	`min.w.t Vk`
4920	`min.w.f Vk`
4921	`min.w.f Vk`
2925	`min.l.t Vk`
2926	`min.l.t Vk`
4925	`min.l.f Vk`
4926	`min.l.f Vk`
2930	`min.s.t Vk` (native mode)
2931	`min.s.t Vk` (native mode)
3930	`min.s.t Vk` (IEEE mode)
3931	`min.s.t Vk` (IEEE mode)
4930	`min.s.f Vk` (native mode)
4931	`min.s.f Vk` (native mode)
5930	`min.s.f Vk` (IEEE mode)
5931	`min.s.f Vk` (IEEE mode)
2935	`min.d.t Vk` (native mode)
2936	`min.d.t Vk` (native mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
3935	`min.d.t Vk` (IEEE mode)
3936	`min.d.t Vk` (IEEE mode)
4935	`min.d.f Vk` (native mode)
4936	`min.d.f Vk` (native mode)
5935	`min.d.f Vk` (IEEE mode)
5936	`min.d.f Vk` (IEEE mode)
2940	`all.t Vk`
2941	`all.t Vk`
4940	`all.f Vk`
4941	`all.f Vk`
2945	`any.t Vk`
2946	`any.t Vk`
4945	`any.f Vk`
4946	`any.f Vk`
2950	`parity.t Vk`
2951	`parity.t Vk`
4950	`parity.f Vk`
4951	`parity.f Vk`
955	`cvtb.w Vj, Vk`
956	`cvtb.w Vj, Vk`
2955	`cvtb.w.t Vj, Vk`
2956	`cvtb.w.t Vj, Vk`
4955	`cvtb.w.f Vj, Vk`
4956	`cvtb.w.f Vj, Vk`
957	`cvth.w Vj, Vk`
958	`cvth.w Vj, Vk`
2957	`cvth.w.t Vj, Vk`
2958	`cvth.w.t Vj, Vk`

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
4957	`cvth.w.f Vj, Vk`
4958	`cvth.w.f Vj, Vk`
960	`cvtw.b Vj, Vk`
961	`cvtw.b Vj, Vk`
2960	`cvtw.b.t Vj, Vk`
2961	`cvtw.b.t Vj, Vk`
4960	`cvtw.b.f Vj, Vk`
4961	`cvtw.b.f Vj, Vk`
962	`cvtw.h Vj, Vk`
963	`cvtw.h Vj, Vk`
2962	`cvtw.h.t Vj, Vk`
2963	`cvtw.h.t Vj, Vk`
4962	`cvtw.h.f Vj, Vk`
4963	`cvtw.h.f Vj, Vk`
965	`cvtw.l Vj, Vk`
966	`cvtw.l Vj, Vk`
2965	`cvtw.l.t Vj, Vk`
2966	`cvtw.l.t Vj, Vk`
4965	`cvtw.l.f Vj, Vk`
4966	`cvtw.l.f Vj, Vk`
967	`cvtw.s Vj, Vk` (native mode)
968	`cvtw.s Vj, Vk` (native mode)
1967	`cvtw.s Vj, Vk` (IEEE mode)
1968	`cvtw.s Vj, Vk` (IEEE mode)
2967	`cvtw.s.t Vj, Vk` (native mode)
2968	`cvtw.s.t Vj, Vk` (native mode)
3967	`cvtw.s.t Vj, Vk` (IEEE mode)
3968	`cvtw.s.t Vj, Vk` (IEEE mode)

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
4967	`cvtw.s.f Vj, Vk` (native mode)
4968	`cvtw.s.f Vj, Vk` (native mode)
5967	`cvtw.s.f Vj, Vk` (IEEE mode)
5968	`cvtw.s.f Vj, Vk` (IEEE mode)
970	`cvtw.d Vj, Vk` (native mode)
971	`cvtw.d Vj, Vk` (native mode)
1970	`cvtw.d Vj, Vk` (IEEE mode)
1971	`cvtw.d Vj, Vk` (IEEE mode)
2970	`cvtw.d.t Vj, Vk` (native mode)
2971	`cvtw.d.t Vj, Vk` (native mode)
3970	`cvtw.d.t Vj, Vk` (IEEE mode)
3971	`cvtw.d.t Vj, Vk` (IEEE mode)
4970	`cvtw.d.f Vj, Vk` (native mode)
4971	`cvtw.d.f Vj, Vk` (native mode)
5970	`cvtw.d.f Vj, Vk` (IEEE mode)
5971	`cvtw.d.f Vj, Vk` (IEEE mode)
975	`cvtl.w Vj, Vk`
976	`cvtl.w Vj, Vk`
2975	`cvtl.w.t Vj, Vk`
2976	`cvtl.w.t Vj, Vk`
4975	`cvtl.w.f Vj, Vk`
4976	`cvtl.w.f Vj, Vk`
977	`cvtl.s Vj, Vk` (native mode)
978	`cvtl.s Vj, Vk` (native mode)
1977	`cvtl.s Vj, Vk` (IEEE mode)
1978	`cvtl.s Vj, Vk` (IEEE mode)
2977	`cvtl.s.t Vj, Vk` (native mode)
2978	`cvtl.s.t Vj, Vk` (native mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
3977	`cvtl.s.t Vj, Vk` (IEEE mode)
3978	`cvtl.s.t Vj, Vk` (IEEE mode)
4977	`cvtl.s.f Vj, Vk` (native mode)
4978	`cvtl.s.f Vj, Vk` (native mode)
5977	`cvtl.s.f Vj, Vk` (IEEE mode)
5978	`cvtl.s.f Vj, Vk` (IEEE mode)
980	`cvtl.d Vj, Vk` (native mode)
981	`cvtl.d Vj, Vk` (native mode)
1980	`cvtl.d Vj, Vk` (IEEE mode)
1981	`cvtl.d Vj, Vk` (IEEE mode)
2980	`cvtl.d.t Vj, Vk` (native mode)
2981	`cvtl.d.t Vj, Vk` (native mode)
3980	`cvtl.d.t Vj, Vk` (IEEE mode)
3981	`cvtl.d.t Vj, Vk` (IEEE mode)
4980	`cvtl.d.f Vj, Vk` (native mode)
4981	`cvtl.d.f Vj, Vk` (native mode)
5980	`cvtl.d.f Vj, Vk` (IEEE mode)
5981	`cvtl.d.f Vj, Vk` (IEEE mode)
985	`cvts.w Vj, Vk` (native mode)
986	`cvts.w Vj, Vk` (native mode)
1985	`cvts.w Vj, Vk` (IEEE mode)
1986	`cvts.w Vj, Vk` (IEEE mode)
2985	`cvts.w.t Vj, Vk` (native mode)
2986	`cvts.w.t Vj, Vk` (native mode)
3985	`cvts.w.t Vj, Vk` (IEEE mode)
3986	`cvts.w.t Vj, Vk` (IEEE mode)
4985	`cvts.w.f Vj, Vk` (native mode)
4986	`cvts.w.f Vj, Vk` (native mode)

Table 109 (continued)
cpu4241 class 2 subtests

Subtest	Instruction tested/test performed
5985	`cvts.w.f Vj, Vk` (IEEE mode)
5986	`cvts.w.f Vj, Vk` (IEEE mode)
987	`cvts.l Vj, Vk` (native mode)
988	`cvts.l Vj, Vk` (native mode)
1987	`cvts.l Vj, Vk` (IEEE mode)
1988	`cvts.l Vj, Vk` (IEEE mode)
2987	`cvts.l.t Vj, Vk` (native mode)
2988	`cvts.l.t Vj, Vk` (native mode)
3987	`cvts.l.t Vj, Vk` (IEEE mode)
3988	`cvts.l.t Vj, Vk` (IEEE mode)
4987	`cvts.l.f Vj, Vk` (native mode)
4988	`cvts.l.f Vj, Vk` (native mode)
5987	`cvts.l.f Vj, Vk` (IEEE mode)
5988	`cvts.l.f Vj, Vk` (IEEE mode)
990	`cvts.d Vj, Vk` (native mode)
991	`cvts.d Vj, Vk` (native mode)
1990	`cvts.d Vj, Vk` (IEEE mode)
1991	`cvts.d Vj, Vk` (IEEE mode)
2990	`cvts.d.t Vj, Vk` (native mode)
2991	`cvts.d.t Vj, Vk` (native mode)
3990	`cvts.d.t Vj, Vk` (IEEE mode)
3991	`cvts.d.t Vj, Vk` (IEEE mode)
4990	`cvts.d.f Vj, Vk` (native mode)
4991	`cvts.d.f Vj, Vk` (native mode)
5990	`cvts.d.f Vj, Vk` (IEEE mode)
5991	`cvts.d.f Vj, Vk` (IEEE mode)
992	`cvtd.w Vj, Vk` (native mode)
993	`cvtd.w Vj, Vk` (native mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
1992	`cvtd.w Vj, Vk` (IEEE mode)
1993	`cvtd.w Vj, Vk` (IEEE mode)
2992	`cvtd.w.t Vj, Vk` (native mode)
2993	`cvtd.w.t Vj, Vk` (native mode)
3992	`cvtd.w.t Vj, Vk` (IEEE mode)
3993	`cvtd.w.t Vj, Vk` (IEEE mode)
4992	`cvtd.w.f Vj, Vk` (native mode)
4993	`cvtd.w.f Vj, Vk` (native mode)
5992	`cvtd.w.f Vj, Vk` (IEEE mode)
5993	`cvtd.w.f Vj, Vk` (IEEE mode)
995	`cvtd.l Vj, Vk` (native mode)
996	`cvtd.l Vj, Vk` (native mode)
1995	`cvtd.l Vj, Vk` (IEEE mode)
1996	`cvtd.l Vj, Vk` (IEEE mode)
2995	`cvtd.l.t Vj, Vk` (native mode)
2996	`cvtd.l.t Vj, Vk` (native mode)
3995	`cvtd.l.t Vj, Vk` (IEEE mode)
3996	`cvtd.l.t Vj, Vk` (IEEE mode)
4995	`cvtd.l.f Vj, Vk` (native mode)
4996	`cvtd.l.f Vj, Vk` (native mode)
5995	`cvtd.l.f Vj, Vk` (IEEE mode)
5996	`cvtd.l.f Vj, Vk` (IEEE mode)
997	`cvtd.s Vj, Vk` (native mode)
998	`cvtd.s Vj, Vk` (native mode)
1997	`cvtd.s Vj, Vk` (IEEE mode)
1998	`cvtd.s Vj, Vk` (IEEE mode)
2997	`cvtd.s.t Vj, Vk` (native mode)
2998	`cvtd.s.t Vj, Vk` (native mode)

Table 109 (continued)
`cpu4241` class 2 subtests

Subtest	Instruction tested/test performed
3997	`cvtd.s.t Vj, Vk` (IEEE mode)
3998	`cvtd.s.t Vj, Vk` (IEEE mode)
4997	`cvtd.s.f Vj, Vk` (native mode)
4998	`cvtd.s.f Vj, Vk` (native mode)
5997	`cvtd.s.f Vj, Vk` (IEEE mode)
5998	`cvtd.s.f Vj, Vk` (IEEE mode)

Class 3 subtests

Class 3 subtests, listed in Table 110, verify the operation of the multiply and divide pipe. Specifically, this class verifies the vector/vector and scalar/vector multiplications and divisions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 110
`cpu4241` class 3 subtests

Subtest	Instruction tested/test performed
2630	`mul.b.t Vi, Sj, Vk`
2631	`mul.b.t Vi, Sj, Vk`
4630	`mul.b.f Vi, Sj, Vk`
4631	`mul.b.f Vi, Sj, Vk`
2635	`mul.h.t Vi, Sj, Vk`
2636	`mul.h.t Vi, Sj, Vk`
4635	`mul.h.f Vi, Sj, Vk`
4636	`mul.h.f Vi, Sj, Vk`
2640	`mul.w.t Vi, Sj, Vk`
2641	`mul.w.t Vi, Sj, Vk`
4640	`mul.w.f Vi, Sj, Vk`
4641	`mul.w.f Vi, Sj, Vk`
2645	`mul.l.t Vi, Sj, Vk`
2646	`mul.l.t Vi, Sj, Vk`
4645	`mul.l.f Vi, Sj, Vk`
4646	`mul.l.f Vi, Sj, Vk`
2650	`mul.s.t Vi, Sj, Vk` (native mode)
2651	`mul.s.t Vi, Sj, Vk` (native mode)
3650	`mul.s.t Vi, Sj, Vk` (IEEE mode)
3651	`mul.s.t Vi, Sj, Vk` (IEEE mode)
4650	`mul.s.f Vi, Sj, Vk` (native mode)

Table 110 (continued)
cpu4241 class 3 subtests

Subtest	Instruction tested/test performed
4651	`mul.s.f Vi, Sj, Vk` (native mode)
5650	`mul.s.f Vi, Sj, Vk` (IEEE mode)
5651	`mul.s.f Vi, Sj, Vk` (IEEE mode)
2655	`mul.d.t Vi, Sj, Vk` (native mode)
2656	`mul.d.t Vi, Sj, Vk` (native mode)
3655	`mul.d.t Vi, Sj, Vk` (IEEE mode)
3656	`mul.d.t Vi, Sj, Vk` (IEEE mode)
4655	`mul.d.f Vi, Sj, Vk` (native mode)
4656	`mul.d.f Vi, Sj, Vk` (native mode)
5655	`mul.d.f Vi, Sj, Vk` (IEEE mode)
5656	`mul.d.f Vi, Sj, Vk` (IEEE mode)
2660	`diVkb.t Vi, Sj, Vk`
2661	`diVkb.t Vi, Sj, Vk`
4660	`diVkb.f Vi, Sj, Vk`
4661	`diVkb.f Vi, Sj, Vk`
2665	`diVkh.t Vi, Sj, Vk`
2666	`diVkh.t Vi, Sj, Vk`
4665	`diVkh.f Vi, Sj, Vk`
4666	`diVkh.f Vi, Sj, Vk`
2670	`diVkw.t Vi, Sj, Vk`
2671	`diVkw.t Vi, Sj, Vk`
4670	`diVkw.f Vi, Sj, Vk`
4671	`diVkw.f Vi, Sj, Vk`
2675	`diVkl.t Vi, Sj, Vk`
2676	`diVkl.t Vi, Sj, Vk`
4675	`diVkl.f Vi, Sj, Vk`
4676	`diVkl.f Vi, Sj, Vk`
2680	`diVks.t Vi, Sj, Vk` (native mode)

Table 110 (continued)
`cpu4241` class 3 subtests

Subtest	Instruction tested/test performed
2681	`diVks.t Vi, Sj, Vk` (native mode)
3680	`diVks.t Vi, Sj, Vk` (IEEE mode)
3681	`diVks.t Vi, Sj, Vk` (IEEE mode)
4680	`diVks.f Vi, Sj, Vk` (native mode)
4681	`diVks.f Vi, Sj, Vk` (native mode)
5680	`diVks.f Vi, Sj, Vk` (IEEE mode)
5681	`diVks.f Vi, Sj, Vk` (IEEE mode)
2685	`diVkd.t Vi, Sj, Vk` (native mode)
2686	`diVkd.t Vi, Sj, Vk` (native mode)
3685	`diVkd.t Vi, Sj, Vk` (IEEE mode)
3686	`diVkd.t Vi, Sj, Vk` (IEEE mode)
4685	`diVkd.f Vi, Sj, Vk` (native mode)
4686	`diVkd.f Vi, Sj, Vk` (native mode)
5685	`diVkd.f Vi, Sj, Vk` (IEEE mode)
5686	`diVkd.f Vi, Sj, Vk` (IEEE mode)
690	`diVks Si, Vj, Vk` (native mode)
691	`diVks Si, Vj, Vk` (native mode)
1690	`diVks Si, Vj, Vk` (IEEE mode)
1691	`diVks Si, Vj, Vk` (IEEE mode)
2690	`diVks.t Si, Vj, Vk` (native mode)
2691	`diVks.t Si, Vj, Vk` (native mode)
3690	`diVks.t Si, Vj, Vk` (IEEE mode)
3691	`diVks.t Si, Vj, Vk` (IEEE mode)
4690	`diVks.f Si, Vj, Vk` (native mode)
4691	`diVks.f Si, Vj, Vk` (native mode)
5690	`diVks.f Si, Vj, Vk` (IEEE mode)
5691	`diVks.f Si, Vj, Vk` (IEEE mode)
695	`diVkd Si, Vj, Vk` (native mode)

Table 110 (continued)
cpu4241 class 3 subtests

Subtest	Instruction tested/test performed
696	`diVkd Si, Vj, Vk` (native mode)
1695	`diVkd Si, Vj, Vk` (IEEE mode)
1696	`diVkd Si, Vj, Vk` (IEEE mode)
2695	`diVkd.t Si, Vj, Vk` (native mode)
2696	`diVkd.t Si, Vj, Vk` (native mode)
3695	`diVkd.t Si, Vj, Vk` (IEEE mode)
3696	`diVkd.t Si, Vj, Vk` (IEEE mode)
4695	`diVkd.f Si, Vj, Vk` (native mode)
4696	`diVkd.f Si, Vj, Vk` (native mode)
5695	`diVkd.f Si, Vj, Vk` (IEEE mode)
5696	`diVkd.f Si, Vj, Vk` (IEEE mode)
2700	`mul.b.t Vi, Vj, Vk`
2701	`mul.b.t Vi, Vj, Vk`
4700	`mul.b.f Vi, Vj, Vk`
4701	`mul.b.f Vi, Vj, Vk`
2705	`mul.h.t Vi, Vj, Vk`
2706	`mul.h.t Vi, Vj, Vk`
4705	`mul.h.f Vi, Vj, Vk`
4706	`mul.h.f Vi, Vj, Vk`
2710	`mul.w.t Vi, Vj, Vk`
2711	`mul.w.t Vi, Vj, Vk`
4710	`mul.w.f Vi, Vj, Vk`
4711	`mul.w.f Vi, Vj, Vk`
2715	`mul.l.t Vi, Vj, Vk`
2716	`mul.l.t Vi, Vj, Vk`
4715	`mul.l.f Vi, Vj, Vk`
4716	`mul.l.f Vi, Vj, Vk`
2720	`mul.s.t Vi, Vj, Vk` (native mode)

Table 110 (continued)
`cpu4241` class 3 subtests

Subtest	Instruction tested/test performed
2721	`mul.s.t Vi, Vj, Vk` (native mode)
3720	`mul.s.t Vi, Vj, Vk` (IEEE mode)
3721	`mul.s.t Vi, Vj, Vk` (IEEE mode)
4720	`mul.s.f Vi, Vj, Vk` (native mode)
4721	`mul.s.f Vi, Vj, Vk` (native mode)
5720	`mul.s.f Vi, Vj, Vk` (IEEE mode)
5721	`mul.s.f Vi, Vj, Vk` (IEEE mode)
2725	`mul.d.t Vi, Vj, Vk` (native mode)
2726	`mul.d.t Vi, Vj, Vk` (native mode)
3725	`mul.d.t Vi, Vj, Vk` (IEEE mode)
3726	`mul.d.t Vi, Vj, Vk` (IEEE mode)
4725	`mul.d.f Vi, Vj, Vk` (native mode)
4726	`mul.d.f Vi, Vj, Vk` (native mode)
5725	`mul.d.f Vi, Vj, Vk` (IEEE mode)
5726	`mul.d.f Vi, Vj, Vk` (IEEE mode)
730	`sqrt.s Vj, Vk` (native mode)
731	`sqrt.s Vj, Vk` (native mode)
1730	`sqrt.s Vj, Vk` (IEEE mode)
1731	`sqrt.s Vj, Vk` (IEEE mode)
2730	`sqrt.s.t Vj, Vk` (native mode)
2731	`sqrt.s.t Vj, Vk` (native mode)
3730	`sqrt.s.t Vj, Vk` (IEEE mode)
3731	`sqrt.s.t Vj, Vk` (IEEE mode)
4730	`sqrt.s.f Vj, Vk` (native mode)
4731	`sqrt.s.f Vj, Vk` (native mode)
5730	`sqrt.s.f Vj, Vk` (IEEE mode)
5731	`sqrt.s.f Vj, Vk` (IEEE mode)
735	`sqrt.d Vj, Vk` (native mode)

Table 110 (continued)
`cpu4241` class 3 subtests

Subtest	Instruction tested/test performed
736	`sqrt.d Vj, Vk` (native mode)
1735	`sqrt.d Vj, Vk` (IEEE mode)
1736	`sqrt.d Vj, Vk` (IEEE mode)
2735	`sqrt.d.t Vj, Vk` (native mode)
2736	`sqrt.d.t Vj, Vk` (native mode)
3735	`sqrt.d.t Vj, Vk` (IEEE mode)
3736	`sqrt.d.t Vj, Vk` (IEEE mode)
4735	`sqrt.d.f Vj, Vk` (native mode)
4736	`sqrt.d.f Vj, Vk` (native mode)
5735	`sqrt.d.f Vj, Vk` (IEEE mode)
5736	`sqrt.d.f Vj, Vk` (IEEE mode)
2740	`diVkb.t Vi, Vj, Vk`
2741	`diVkb.t Vi, Vj, Vk`
4740	`diVkb.f Vi, Vj, Vk`
4741	`diVkb.f Vi, Vj, Vk`
2745	`diVkh.t Vi, Vj, Vk`
2746	`diVkh.t Vi, Vj, Vk`
4745	`diVkh.f Vi, Vj, Vk`
4746	`diVkh.f Vi, Vj, Vk`
2750	`diVkw.t Vi, Vj, Vk`
2751	`diVkw.t Vi, Vj, Vk`
4750	`diVkw.f Vi, Vj, Vk`
4751	`diVkw.f Vi, Vj, Vk`
2755	`diVkl.t Vi, Vj, Vk`
2756	`diVkl.t Vi, Vj, Vk`
4755	`diVkl.f Vi, Vj, Vk`
4756	`diVkl.f Vi, Vj, Vk`
2760	`diVks.t Vi, Vj, Vk` (native mode)

Table 110 (continued)
`cpu4241` class 3 subtests

Subtest	Instruction tested/test performed
2761	`diVks.t Vi, Vj, Vk` (native mode)
3760	`diVks.t Vi, Vj, Vk` (IEEE mode)
3761	`diVks.t Vi, Vj, Vk` (IEEE mode)
4760	`diVks.f Vi, Vj, Vk` (native mode)
4761	`diVks.f Vi, Vj, Vk` (native mode)
5760	`diVks.f Vi, Vj, Vk` (IEEE mode)
5761	`diVks.f Vi, Vj, Vk` (IEEE mode)
2765	`diVkd.t Vi, Vj, Vk` (native mode)
2766	`diVkd.t Vi, Vj, Vk` (native mode)
3765	`diVkd.t Vi, Vj, Vk` (IEEE mode)
3766	`diVkd.t Vi, Vj, Vk` (IEEE mode)
4765	`diVkd.f Vi, Vj, Vk` (native mode)
4766	`diVkd.f Vi, Vj, Vk` (native mode)
5765	`diVkd.f Vi, Vj, Vk` (IEEE mode)
5766	`diVkd.f Vi, Vj, Vk` (IEEE mode)
2850	`prod.b.t Vk`
2851	`prod.b.t Vk`
4850	`prod.b.f Vk`
4851	`prod.b.f Vk`
2855	`prod.h.t Vk`
2856	`prod.h.t Vk`
4855	`prod.h.f Vk`
4856	`prod.h.f Vk`
2860	`prod.w.t Vk`
2861	`prod.w.t Vk`
4860	`prod.w.f Vk`
4861	`prod.w.f Vk`
2865	`prod.l.t Vk`

Table 110 (continued)
cpu4241 class 3 subtests

Subtest	Instruction tested/test performed
2866	`prod.l.t Vk`
4865	`prod.l.f Vk`
4866	`prod.l.f Vk`
2870	`prod.s.t Vk` (native mode)
2871	`prod.s.t Vk` (native mode)
3870	`prod.s.t Vk` (IEEE mode)
3871	`prod.s.t Vk` (IEEE mode)
4870	`prod.s.f Vk` (native mode)
4871	`prod.s.f Vk` (native mode)
5870	`prod.s.f Vk` (IEEE mode)
5871	`prod.s.f Vk` (IEEE mode)
2875	`prod.d.t Vk` (native mode)
2876	`prod.d.t Vk` (native mode)
3875	`prod.d.t Vk` (IEEE mode)
3876	`prod.d.t Vk` (IEEE mode)
4875	`prod.d.f Vk` (native mode)
4876	`prod.d.f Vk` (native mode)
5875	`prod.d.f Vk` (IEEE mode)
5876	`prod.d.f Vk` (IEEE mode)

Class 4 subtests

Class 4 subtests (see Table 111) verify the operation of loading and storing vector registers. Specifically, this class verifies loading and storing the vector using direct addressing and vector of indices, and storing of vectors and scalar registers using the extended operations.

All subtests end with a `halt` command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 111
`cpu4241` class 4 subtests

Subtest	Instruction tested/test performed
2200	`ld.b.t` *effa*, `Vk`
2201	`ld.b.t` *effa*, `Vk`
4200	`ld.b.f` *effa*, `Vk`
4201	`ld.b.f` *effa*, `Vk`
2205	`ld.h.t` *effa*, `Vk`
2206	`ld.h.t` *effa*, `Vk`
4205	`ld.h.f` *effa*, `Vk`
4206	`ld.h.f`*effa*, `Vk`
2210	`ld.w.t` *effa*, `Vk`
2211	`ld.w.t` *effa*, `Vk`
4210	`ld.w.f` *effa*, `Vk`
4211	`ld.w.f`*effa*, `Vk`
2215	`ld.l.t` *effa*, `Vk`
2216	`ld.l.t` *effa*, `Vk`
4215	`ld.l.f` *effa*, `Vk`
4216	`ld.l.f`*effa*, `Vk`
2230	`st.b.t Vk,` *effa*
2231	`st.b.t Vk,` *effa*
4230	`st.b.f Vk,` *effa*
4231	`st.b.f Vk,` *effa*

Table 111 (continued)
`cpu4241` class 4 subtests

Subtest	Instruction tested/test performed
2235	`st.h.t Vk, effa`
2236	`st.h.t Vk, effa`
4235	`st.h.f Vk, effa`
4236	`st.h.f Vk, effa`
2240	`st.w.t Vk, effa`
2241	`st.w.t Vk, effa`
4240	`st.w.f Vk, effa`
4241	`st.w.f Vk, effa`
2245	`st.l.t Vk, effa`
2246	`st.l.t Vk, effa`
4245	`st.l.f Vk, effa`
4246	`st.l.f Vk, effa`
2550	`ldvi.b.t Vj, Vk`
2551	`ldvi.b.t Vj, Vk`
4550	`ldvi.b.f Vj, Vk`
4551	`ldvi.b.f Vj, Vk`
2555	`ldvi.h.t Vj, Vk`
2556	`ldvi.h.t Vj, Vk`
4555	`ldvi.h.f Vj, Vk`
4556	`ldvi.h.f Vj, Vk`
2560	`ldvi.w.t Vj, Vk`
2561	`ldvi.w.t Vj, Vk`
4560	`ldvi.w.f Vj, Vk`
4561	`ldvi.w.f Vj, Vk`
2565	`ldvi.l.t Vj, Vk`
2566	`ldvi.l.t Vj, Vk`
4565	`ldvi.l.f Vj, Vk`
4566	`ldvi.l.f Vj, Vk`

Table 111 (continued)
cpu42 41 class 4 subtests

Subtest	Instruction tested/test performed
2570	stvi.b.t Vi, Vj
2571	stvi.b.t Vi, Vj
4570	stvi.b.f Vi, Vj
4571	stvi.b.f Vi, Vj
2575	stvi.h.t Vi, Vj
2576	stvi.h.t Vi, Vj
4575	stvi.h.f Vi, Vj
4576	stvi.h.f Vi, Vj
2580	stvi.w.t Vi, Vj
2581	stvi.w.t Vi, Vj
4580	stvi.w.f Vi, Vj
4581	stvi.w.f Vi, Vj
2585	stvi.l.t Vi, Vj
2586	stvi.l.t Vi, Vj
4585	stvi.l.f Vi, Vj
4586	stvi.l.f Vi, Vj
2590	stvi.b.t Si, Vj
2591	stvi.b.t Si, Vj
4590	stvi.b.f Si, Vj
4591	stvi.b.f Si, Vj
2595	stvi.h.t Si, Vj
2596	stvi.h.t Si, Vj
4595	stvi.h.f Si, Vj
4596	stvi.h.f Si, Vj
2600	stvi.w.t Si, Vj
2601	stvi.w.t Si, Vj
4600	stvi.w.f Si, Vj
4601	stvi.w.f Si, Vj

Table 111 (continued)
`cpu4241` class 4 subtests

Subtest	Instruction tested/test performed
2605	`stvi.l.t Si, Vj`
2606	`stvi.l.t Si, Vj`
4605	`stvi.l.f Si, Vj`
4606	`stvi.l.f Si, Vj`
2610	`ste.b.t Sk,` *effa*
2611	`ste.b.t Sk,` *effa*
4610	`ste.b.f Sk,` *effa*
4611	`ste.b.f Sk,` *effa*
2615	`ste.h.t Sk,` *effa*
2616	`ste.h.t Sk,` *effa*
4615	`ste.h.f Sk,` *effa*
4616	`ste.h.f Sk,` *effa*
2620	`ste.w.t Vk,` *effa*
2621	`ste.w.t Vk,` *effa*
4620	`ste.w.f Vk,` *effa*
4621	`ste.w.f Vk,` *effa*
2625	`ste.l.t Vk,` *effa*
2626	`ste.l.t Vk,` *effa*
4625	`ste.l.f Vk,` *effa*
4626	`ste.l.f Vk,` *effa*

Privileged instructions and architectural features (cpu4331)

14

The `cpu4331` test verifies nonvector, single-headed architectural features unique to the C3400 Series processors. Included are tests of system exceptions, interrupts, privileged instructions, the various processor caches, remote invalidates, and nonresident memory pages.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 112.

Table 112
`cpu4331` functional areas tested

Functional area	Tested by the diagnostic	Exercised by the diagnostic
CPU	Yes	No
CUJ	Yes	No
Memory even	No	Yes
Memory odd	No	Yes
PI2	No	Yes
CCU	No	No
SP5	Yes	No

In order to run the `cpu4331` test, the boards listed in Table 113 must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table 113
`cpu4331` required functional boards

Board	Test to verify
Service processor (SP5)	`spu1000, spu4000`
Memory system (MCM, MCM2, MCM3)	`mem4100`
Central processor unit (CPU)	`cpu4030`
CPU utility board (CUJ)	`SCAN_CUJ`
PBUS interface adapter (PI2)	`pi2_4000`

Note

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the `cpu4331` test, use the procedure shown in Figure 171.

Figure 171
`cpu4331` test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL
: test cpu4331 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if `mem4100` was executed with failures, or if `spu4000` was executed, then `initall` must be executed prior to test execution, but only after the SPU EPROM based self-test has passed. Failure to execute `initall` in these circumstances could result in invalid test results.

Entering only

```
test cpu4331
```

executes all `cpu4331` subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the `-c` or `-s` options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 172.

Figure 172
`cpu4331` test parameter menu

```
Test 'cpu4331.t'                              Thu Nov 19 00:00:00 1965

                    ENTER TEST PARAMETERS
       [ ]    Encloses allowed input ranges or values
       ( )    Encloses the default value
       ^      Returns to the previous prompt
       :nn    Returns to the prompt # nn
       :      Returns to the first unsatisfied prompt
       :?     Reviews previous entries

1. Run default switches? [y,n]                                  (y) ->
2. CPUs to test: [01234567]                                (01234567) ->
3. Forced Faulting Enabled? [y,n]                               (n) ->
4. Fault on Instruction Fetches? [y,n]                          (n) ->
5. Sequential Execution? [y,n]                                  (n) ->
6. Timeout Scale Factor Enabled? [1-100]                        (1) ->
7. Dcache Enabled? [y,n]                                        (y) ->
8. Segment of Execution? [0-7]                                  (0) ->
9: Chained Execution Mode? [y,n]                                (n) ->
10: Loop Enabled? [y,n]                                         (n) ->
11: Hard Errors Enabled? [y,n]                                  (y) ->
12: Load CPU Code? [y,n]                                        (y) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

In Figure 172, prompt 2, `[01234567]` represents all available CPUs in the machine under test.

Prompt explanations

A description of each prompt and its meaning follows:

```
1: Run default switches? [y/n]                (y) ->
```

If you respond with **y** or **RETURN**, no additional test parameter prompts are displayed and testing begins.

However, if you respond with **n**, additional test parameter prompts are displayed, allowing choices other than the default selections. The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
2: CPUs to test: [01234567]                   (01234567) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selections, and the default, represented by `01234567`, consist of all available CPUs.

```
3: Forced Faulting Enabled? [y,n]             (n) ->
```

If you answer with **y**, normal force faulting occurs on all data references: the system will force a nonresident data exception to occur on every data reference.

If this option is enabled, subtest execution time is increased and the timeout scale factor requires adjustment to prevent the SPU from terminating the test prematurely.

```
4: Fault on Instruction Fetches? [y,n]        (n)->
```

If you answer with **y**, force faulting occurs on instruction fetches, in addition to data references. This prompt appears only if the previous prompt is answered **y**.

```
5: Sequential Execution? [y,n]          (n) ->
```

If you answer with **y**, the sequential bit in the processor status word (PSW) is set to forced sequential execution mode.

```
6: Timeout Scale Factor [1-100]               (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
7: Dcache Enabled? [y,n]        (y) ->
```

The Dcache is normally enabled. However, if it is suspected to be broken, it can be disabled by entering **n** at this prompt.

```
8: Segment of Execution? [0-7] (0) ->
```

The segment of execution is contained in bits <31..29> of the program counter (PC).

- If **0** is entered, then bits <31..29> of the PC are `000` and the test is run in ring zero.
- If **1** is entered, then bits <31..29> of the PC are `001` and the test is run in ring one.
- If **2** is entered, then bits <31..29> of the PC are `010` and the test is run in ring two.
- If **3** is entered, then bits <31..29> of the PC are `011` and the test is run in ring three.
- If **4, 5, 6,** or **7** is entered, then bits <31..29> of the PC are `100, 101, 110,` and `111` respectively, and the test is run in ring four.

Refer to the *CONVEX Architecture Reference Manual (C Series)* for more information concerning the meaning of rings in the machine architecture.

```
9: Chained Execution Mode? [y,n]      (n) ->
```

If this option is enabled by entering **y**, the test is executed in chained mode, which causes the CPU to perform subtest sequencing. The SPU is unaware of the action of the CPU, unless a subtest fails or unless all of the subtests pass, so no subtest execution tracing occurs in this mode. All subtests are executed, however, and test execution time is greatly reduced. Also, this option cannot be enabled if the `-c` or the `-s` options were used in the invocation procedure.

```
10: Loop Enabled? [y,n]         (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by pressing **CTRL-C**.

```
11: Hard Errors Enabled? [y,n] (y) ->
```

If this option is enabled and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled, parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
12: Load CPU Code? [y,n]        (y) ->
```

If the CPU code for this test is already in memory, you can enter n for this prompt and the code will not be reloaded (thus saving time).

Test parameter summary

When all prompts have been answered, a test parameter summary echoes the prompts that have been answered. Figure 173 displays a sample test parameter summary.

Figure 173
cpu4331 test parameter summary

```
                TEST PARAMETER SUMMARY

Run default switches?                          : N
Cpus to test:                                  : 01
Forced Faulting Enabled?                       : y
Sequential Execution?                          : n
Timeout Scale Factor Enabled?                  : 1
Dcache Enabled?                                : y
Segment of Execution?                          : 0
Chained Execution Mode?                        : n
Loop Enabled?                                  : n
Hard Errors Enabled?                           : y
Load CPU Code?                                 : y
```

The actual summary varies depending on the answers to the prompts.

Hardware initialization sequence

After the last prompt is entered, and before test code execution, the following events occur:

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CUJ, PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- Clocks are turned on for each selected CPU, one at a time.

Note

The first two events are accomplished at the initial start of the `cpu4331` test. The remaining events are accomplished when each subtest is initialized.

Current memory allocation

Immediately before test code execution, a current memory allocation screen is displayed. Figure 174 is an example of the current memory allocation screen.

Figure 174
cpu4331 current memory allocation screen

```
                    Current Memory Allocation

File No.  Physical Address Pid File Name                Logical Offset
      1   00000000-00027fff   0 p0r0_4331               00000000
      2   00028000-000b1fff   0 cpu4331.rnn             00022000
      1   000b2000-000d9fff   1 p0r0_4331               00000000
      2   000da000-00163fff   1 cpu4331.rnn             00022000
          03ff7000-03ff9fff   1 ptet   NA
          03ffa000-03ffafff   1 pte2   NA
          03ffb000-03ffdfff   0 ptet   NA
          03ffe000-03ffefff   0 pte2   NA
          03fffc00-3fffffff   1 pte1   NA
```

The physical and logical addresses shown, as well as the file names, are only representative. The actual addresses will vary depending on installed memory and file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First column**—File number.
- **Second column**—Physical memory addresses where the specified file is loaded (useful in conjunction with the mm(1d) utility).
- **Third column**—Process identification.
- **Fourth column**—File name. (The actual path is */filename* or /mnt/test/CPU/*filename*. The entries `pte1`, `pte2`, and `ptet` are not actual files, but are indications of the page tables.)
- **Fifth column**—Logical starting address of the specified file.

Subtest descriptions

There are four classes of subtests for `cpu4331`.

In the tables in this section, the "Instructions tested/test performed" column lists either a description of what is tested or the instruction that is tested. For more information on individual instructions, refer to the *CONVEX Assembly Language Reference Manual (C Series)*. The object module (the executable code) for all classes is cpu4331.rnn.

Class 1 subtests

Class 1 subtests, as listed in Table 114, verify various instructions and features. Included are tests of system calls, C3400 Series-specific nonvector instructions, thread-level addressing, exceptions, interval timers, and privileged instructions.

Table 114
cpu4331 class 1 subtests

Subtest	Instruction tested/test performed
10	System calls
20	`ldpa aj,ak`
21	`patu` instruction test
22	`pate` instruction test
23	Thread addressing
25	Pich instruction test
30	Traps
31	Deadlocks
32	Unimplemented op codes
35	Exceptions
36	Invalid communication register addresses
37	Trap instructions test
40	Interrupts
41	Interval timer test
42	Timer ring cross check
43	Timer CIR switch check
45	Privileged instruction check
600	Test vector valid test
601	`mov sk,vv`
602	Test non-vector valid trapping just after setting VV
604	`mov toc,sk`
700	Ring crossings with trap bits set
701	Ring crossings with pbkpt bits set

Class 2 subtests

Class 2 subtests, as listed in Table 115, verify proper operation of page faults and nonresident calls, returns, and instructions. Class 2 subtests also verify proper operation of subroutine calls and returns for cases where the code and/or the stack are in nonresident memory.

Table 115
cpu4331 class 2 subtests

Subtest	Instruction tested/test performed
50	`callq (abs/@)` into nonresident page
51	`rtnq` into nonresident page
52	`callq` with nonresident stack
53	`rtnq` with nonresident stack
60	`call (abs/@)` into nonresident page
61	`rtn` into nonresident page
62	`call (abs/@)` with nonresident stack
63	`rtn` with nonresident stack
70	`calls (abs/@)` into nonresident page
71	`rtn` into nonresident page
72	`calls` with nonresident stack
73	`rtn` with nonresident stack
80	Nonresident stack, nonresident target page (`callq`)
81	Nonresident stack, nonresident target page (`calls`)
82	Nonresident stack, nonresident target page (`call`)
85	Indirect nonresident, nonresident target page (`callq`)
86	Indirect nonresident, nonresident target page (`calls`)
87	Indirect nonresident, nonresident target page (`call`)
90	Indirect nonresident, nonresident stack (`callq`)
91	Indirect nonresident, nonresident stack (`calls`)

Table 115 (continued)
cpu4331 class 2 subtests

Subtest	Instruction tested/test performed
92	Indirect nonresident, nonresident stack (`call`)
95	Nonresident stack, nonresident return page (`callq`)
96	Nonresident stack, nonresident return page (`calls`)
97	Nonresident stack, nonresident return page (`call`)
100	Nonresident stack, nonresident target page, nonresident indirect address (`callq`)
101	Nonresident stack, nonresident target page, nonresident indirect address (`calls`)
102	Nonresident stack, nonresident target page, nonresident indirect address (`call`)
200	Halfword, nonresident page execute
201	Word, nonresident page execute
202	3-halfword, nonresident page execute
322	`ip` look-ahead faults

Class 3 subtests

Class 3 subtests listed in Table 116, verify proper operation of memory operations (loads and stores) for various conditions (byte, halfword, word, longword, or nonresident).

Table 116
cpu4331 class 3 subtests

Subtest	Instruction tested/test performed
210	Byte loads nonresident page
220	Load halfword nonresident page
221	Load halfword nonresident page
222	Load halfword nonresident page
223	Load halfword nonresident page
224	Load halfword nonresident page
225	Load halfword nonresident page
226	Load halfword nonresident page
227	Load halfword nonresident page
228	Load halfword nonresident page
230	Load word nonresident page
231	Load word nonresident page
232	Load word nonresident page
233	Load word nonresident page
234	Load word nonresident page
235	Load word nonresident page
236	Load word nonresident page
237	Load word nonresident page
238	Load word nonresident page
240	Load longword nonresident page
241	Load longword nonresident page
242	Load longword nonresident page
243	Load longword nonresident page
244	Load longword nonresident page

Table 116 (continued)
cpu4331 class 3 subtests

Subtest	Instruction tested/test performed
245	Load longword nonresident page
246	Load longword nonresident page
247	Load longword nonresident page
248	Load longword nonresident page
250	Store byte nonresident page
260	Store halfword nonresident page
261	Store halfword nonresident page
270	Store word nonresident page
271	Store word nonresident page
272	Store word nonresident page
273	Store word nonresident page
274	Store word nonresident page
275	Store word nonresident page
276	Store word nonresident page
277	Store word nonresident page
278	Store word nonresident page
280	Store longword nonresident page
281	Store longword nonresident page
282	Store longword nonresident page
283	Store longword nonresident page
284	Store longword nonresident page
285	Store longword nonresident page
286	Store longword nonresident page
287	Store longword nonresident page
288	Store longword nonresident page
290	Load byte indirect, address nonresident
291	Load halfword indirect, address nonresident
292	Load word indirect, address nonresident

Table 116 (continued)
cpu4331 class 3 subtests

Subtest	Instruction tested/test performed
293	Load longword indirect, address nonresident
300	Store byte indirect, address nonresident
301	Store halfword indirect, address nonresident
302	Store word indirect, address nonresident
303	Store longword indirect, address nonresident
305	Load byte indirect, address and data nonresident
306	Load byte indirect, address and data nonresident
307	Load byte indirect, address and data nonresident
308	Load byte indirect, address and data nonresident
310	Store byte indirect, address and data nonresident
311	Store byte indirect, address and data nonresident
312	Store byte indirect, address and data nonresident
313	Store byte indirect, address and data nonresident
320	Multiple fault test
323	Page fault combination test I
324	Page fault combination test II
325	Page fault combination test III
326	Page fault combination test IV

Class 4 subtests

Class 4 subtests, as listed in Table 117, verify the proper operation of the various caches in the C3400 Series machines (pte cache, instruction cache, and data cache). These subtests also test remote invalidates between processors.

Table 117
cpu4331 class 4 subtests

Subtest	Instruction tested/test performed
400	Pte cache test
500	Icache test I
501	Icache test II
502	Icache test III
503	Icache test IV
504	Icache test V
505	Icache test VI
506	Icache test VII
510	Dcache test
511	Dcache test
512	Dcache test
513	Dcache test
514	Dcache test
515	Dcache test
516	Dcache test
517	Dcache test
518	Dcache test
519	Dcache test
560	Remote invalidate I
565	Remote invalidate I
570	Remote invalidate I

Enhanced, nonvector, uniprocessor instructions (cpu4332)

15

The `cpu4332` test is an extension of the building block test `cpu4030`. This test provides exhaustive, single CPU testing of the C3400 Series scalar, address, and communication register instructions.

Testing begins by first verifying the operation of the instructions which manipulate the communication registers. The locking/unlocking, putting/getting, sending/receiving, matching/testing, and incrementing of all eight communication register sets are verified to function correctly.

Next, the sending/receiving, pushing/popping, matching/incrementing and test-and-clearing operations on memory are verified.

Third, the scalar instructions (scalar converts; loading/storing of the communication index register (CIR), thread id register (TID), thread timer register (TTR); and loading of the central processing unit identification (CPUID) are verified.

Fourth, the multiprocessor instructions pfork, cfork, wfork, spawn, join, and idle are verified.

Lastly, the remaining instructions trap, pbkpt, ldcmr, stcmr, ctrsl, and ctrsg are verified.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 118.

Table 118
`cpu4332` functional areas tested

Functional area	Tested by diagnostic	Exercised by the diagnostic
CPU	Yes	No
CUJ	No	Yes
Memory even	Yes	No
Memory odd	Yes	No
PI2	No	Yes
CCU	No	No
SP5	No	Yes

In order to run the `cpu4332` test, the boards listed in Table 119 must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table 119
`cpu4332` required functional boards

Board	Test to verify
Service processor (SP5)	`spu1000, spu4000`
Memory system (MCM, MCM2, MCM3)	`mem4100`
CPU utility board (CUJ)	`CUJ_SCAN`
Central processor unit (CPU)	`cpu4030`

Note

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the cpu4332 test, use the procedure shown in Figure 175.

Figure 175
cpu4332 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL
: test cpu4232 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if `mem4100` was executed with failures, or if `spu4000` was executed, then `initall` must be executed prior to test execution, but only after the SPU EPROM based self-test has passed. Failure to execute `initall` in these circumstances could result in invalid test results.

Entering only

```
test cpu4332
```

executes all cpu4332 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the -c or -s options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 176.

Figure 176
`cpu4332` test parameter menu

```
Test 'cpu4332.t'                              Thu Nov 19 00:00:00 1985

                    ENTER TEST PARAMETERS
       [ ]   Encloses allowed input ranges or values
       ( )   Encloses the default value
       . .   Returns to the previous prompt
       :nn   Returns to the prompt # nn
       :     Returns to the first unsatisfied prompt
       :?    Reviews previous entries

1. Run default switches? [y,n]                                  (y) ->
2. CPUs to test: [01234567]                                (01234567) ->
3. Forced Faulting Enabled? [y,n]                               (n) ->
4. Fault on Instruction Fetches? [y,n]                          (n) ->
5. Sequential Execution? [y,n]                                  (n) ->
6. Timeout Scale Factor Enabled? [1-100]                        (1) ->
7. Dcache Enabled? [y,n]                                        (y) ->
8. Segment of Execution? [0-7]                                  (0) ->
9: Chained Execution Mode? [y,n]                                (n) ->
10: Loop Enabled? [y,n]                                         (n) ->
11: Hard Errors Enabled? [y,n]                                  (y) ->
12: Load CPU Code? [y,n]                                        (y) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

In Figure 176, prompt 2, `[01234567]` represents all available CPUs in the machine under test.

Prompt explanations

A description of each prompt and its meaning follows.

```
1: Run default switches? [y/n]                (y) ->
```

If you respond with **y** or **RETURN**, no additional test parameter prompts are displayed and testing begins.

However, if you respond with **n**, additional test parameter prompts are displayed allowing, choices other than the default selections. The following prompts are displayed and answered only if the first prompt is answered with **n**:

```
2: Cpus to test: [01234567]                   (01234567) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selections, and the default, represented by `01234567`, consist of all available CPUs.

```
3: Forced Faulting Enabled? [y,n]             (n) ->
```

If you answer with **y**, normal force faulting occurs on all data references: the system forces a nonresident data exception to occur on every data reference.

If this option is enabled, subtest execution time is increased and the timeout scale factor requires adjustment to prevent the SPU from terminating the test prematurely.

```
4: Fault on Instruction Fetches? [y,n]        (n) ->
```

If you answer with **y**, force faulting occurs on instruction fetches, in addition to data references. This prompt appears only if the previous prompt is answered with **y**.

```
5: Sequential Execution? [y,n]                (n) ->
```

If you answer with **y**, the sequential bit in the processor status word (PSW) is set to forced sequential execution mode.

```
6: Timeout Scale Factor Enabled [1-100]       (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number entered to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it takes the test five times as long to timeout.

```
7: Dcache Enabled? [y,n]                    (y) ->
```

The Dcache is normally enabled. However, if it is suspected to be broken, it can be disabled by entering **n** at this prompt.

```
8: Segment of Execution? [0-7]              (0) ->
```

This prompt is only displayed if forced faults are not enabled. The segment of execution is contained in bits <31..29> of the program counter (PC).

- If **0** is entered, then bits <31..29> of the PC are `000` and the test is run in ring zero.
- If **1** is entered, then bits <31..29> of the PC are `001` and the test is run in ring one.
- If **2** is entered, then bits <31..29> of the PC are `010` and the test is run in ring two.
- If **3** is entered, then bits <31..29> of the PC are `011` and the test is run in ring three.
- If **4, 5, 6,** or **7** is entered, then bits <31..29> of the PC are `100, 101, 110,` and `111`, respectively, and the test is run in ring four.

Refer to the *CONVEX Architecture Reference Manual (C Series)* for more information about the meaning of rings in the machine architecture.

```
9: Chained Execution Mode? [y,n]            (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The service processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time is greatly reduced. The only information printed to the console upon completion is a pass or fail message. Also, this option cannot be enabled if the `-c` or the `-s` options were used in the invocation procedure.

```
10: Loop Enabled? [y,n]                     (n) ->
```

If you answer with **y**, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to branch back to the beginning of the subtest. This puts the subtest into an infinite loop which can be interrupted by pressing **CTRL-C**.

```
11: Hard Errors Enabled? [y,n]                (y) ->
```

If this option is enabled by entering **y**, and a hard error occurs, the clocks are stopped, and the test fails. If this option is disabled by entering **n**, parity errors and other sources of hard errors go undetected. It is recommended that hard errors normally be enabled.

```
12: Load CPU Code? [y,n]                      (y) ->
```

If the CPU code for this test is already in memory, enter **n** for this prompt and the code is not reloaded, thus saving time.

Test parameter summary

When all prompts have been answered, the screen displays a test parameter summary which echoes the prompts that have been answered, as illustrated in Figure 177.

Figure 177
cpu4332 test parameter summary

```
                TEST PARAMETER SUMMARY

Run default switches?                          : N
Cpus to test:                                  : 01
Forced Faulting Enabled?                       : y
Sequential Execution?                          : n
Timeout Scale Factor Enabled?                  : 1
Dcache Enabled?                                : y
Segment of Execution?                          : 0
Chained Execution Mode?                        : n
Loop Enabled?                                  : n
Hard Errors Enabled?                           : y
Load CPU Code?                                 : y
```

The actual summary varies depending on the answers to the prompts.

Hardware initialization sequence

After the last prompt is entered, and before test code execution, the following events occur:

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CUJ, PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- Clocks are turned on for the processor(s) selected. If parallel execution is selected, all clocks are turned on. If sequential execution is selected, each CPU's clocks are turned on one at a time, in the order the CPUs are selected.

Note

The first two events are accomplished at the initial start of the `cpu4332` test. The remaining events are accomplished when each subtest is initialized.

Current memory allocation

Immediately before test code execution, a current memory allocation screen, as illustrated in Figure 178.

Figure 178
cpu4332 current memory allocation screen

```
                    Current Memory Allocation

File No. Physical Address Pid File Name                Logical Offset
      1  00000000-00027fff  0 p0r0_4332               00000000
      2  00028000-000b1fff  0 cpu4332.rnn             00022000
      1  000b2000-000d9fff  1 p0r0_4332               00000000
      2  000da000-00163fff  1 cpu4332.rnn             00022000
         03ff7000-03ff9fff  1 ptet   NA
         03ffa000-03ffafff  1 pte2   NA
         03ffb000-03ffdfff  0 ptet   NA
         03ffe000-03ffefff  0 pte2   NA
         03fffc00-3fffffff  1 pte1   NA
```

The physical and logical addresses shown, as well as the file names, are only representative; the actual addresses will vary depending on installed memory and file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First column**—File number.
- **Second column**—Physical memory addresses where the specified file is loaded (useful in conjunction with the mm(1d) utility).
- **Third column**—Process identification.
- **Fourth column**—File name. (The actual path is */filename* or /mnt/test/CPU/*filename*. The entries `pte1`, `pte2`, and `ptet` are not actual files, but are indications of the page tables.)
- **Fifth column**—Logical starting address of the specified file.

Subtest descriptions

There are five classes of subtests for `cpu4332`.

In the tables in this section, the "Instructions tested" column lists each instruction that is tested. For more information on individual instructions, refer to the *CONVEX Assembly Language Reference Manual (C Series)*. The object module (the executable code) for all classes is cpu4232.rnn.

Class 1 subtests

Class 1 subtests, as listed in Table 120, verify the ability of a single CPU to correctly manipulate each of the testable communication registers. For an explanation of valid manipulations on the communication registers, refer to the *CONVEX Architecture Reference Manual (C Series)*.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 120
Class 1 subtests

Subtest	Instruction tested
100	`lck <Ceffa>`
105	`ulk <Ceffa>`
110	`put.w Ak, <Ceffa>`
111	`putr.w Ak, <effa>`
115	`put.l Sk, <Ceffa>`
116	`putr.l Sk, <effa>`
120	`get.w <Ceffa>, Ak`
121	`getr.w <effa>, Ak`
125	`get.l <Ceffa>, Sk`
126	`getr.l <effa>, Sk`
130	`snd.w Ak, <Ceffa>`
135	`snd.l Sk, <Ceffa>`
140	`rcv.w <Ceffa>, Ak`
145	`rcv.l <Ceffa>, Sk`
150	`mat.w Ak, <Ceffa>`
151	`matr.w Ak, <Ceffa>`
155	`mat.l Sk, <Ceffa>`
156	`matr.l Sk, <Ceffa>`
170	`tst <Ceffa>`
180	`inc.w <Ceffa>, Ak`
185	`inc.l <Ceffa>, Sk`

Class 2 subtests

Class 2 subtests, as listed in Table 121, verifies the instructions correctly manipulate the C3400 Series memory structures, in a single CPU environment. For an explanation of memory structures and how they can be manipulated, refer to the *CONVEX Architecture Reference Manual (C Series).*

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 121
Class 2 subtests

Subtest	Instruction tested
131	`sndr.w Ak, <effa>`
136	`sndr.l Sk, <effa>`
141	`rcvr.w <effa>, Ak`
146	`rcvr.l <effa>, Sk`
160	`matm.w Ak, <effa>`
165	`matm.l Sk, <effa>`
175	`tac <effa>`
190	`pshr <effa>, Ak`
195	`popr Ak, <effa>`
200	`incr.w <effa>, Ak`
205	`incr.l <effa>, Sk`

Class 3 subtests

Class 3 subtests, as listed in Table 122, verify the execution of the new C3400 Series scalar instructions in a single CPU environment. For a detailed explanation of each of the new scalar instructions, refer to the *CONVEX Architecture Reference Manual (C Series)*.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 122
Class 3 subtests

Subtest	Instruction tested
211	`mov Sk, TTR`
212	`mov TTR, Sk`
215	`mov CPUID, Sk`
225	`mov TID, Sk`
230	`mov Sk, TID`
290	`mov Sk, Cir`
295	`mov Cir, Sk`
305	`ldea <effa>, Sk`
310	`shf.w Sj, Sk`
320	`casr`
400	`cvtd.w Sj, Sk` (native mode)
401	`cvtw.d Sj, Sk` (native mode)
410	`cvtd.w Sj, Sk` (IEEE mode)
411	`cvtw.d Sj, Sk` (IEEE mode)
420	`frint.s Sj, Sk` (native mode)
425	`frint.d Sj, Sk` (native mode)
500	`sqrt.s Sk` (native mode)
510	`atan.s Sk` (native mode)
520	`exp.s Sk` (native mode)
530	`ln.s Sk` (native mode)

Table 122 (continued)
Class 3 subtests

Subtest	Instruction tested
540	`sin.s Sk` (native mode)
550	`cos.s Sk` (native mode)
600	`sqrt.d Sk` (native mode)
610	`atan.d Sk` (native mode)
620	`exp.d Sk` (native mode)
630	`ln.d Sk` (native mode)
640	`sin.d Sk` (native mode)
650	`cos.d Sk` (native mode)
1420	`frint.s Sj, Sk` (IEEE mode)
1425	`frint.d Sj, Sk` (IEEE mode)
1500	`sqrt.s Sk` (IEEE mode)
1510	`atan.s Sk` (IEEE mode)
1520	`exp.s Sk` (IEEE mode)
1530	`ln.s Sk` (IEEE mode)
1540	`sin.s Sk` (IEEE mode)
1550	`cos.s Sk` (IEEE mode)
1600	`sqrt.d Sk` (IEEE mode)
1610	`atan.d Sk` (IEEE mode)
1620	`exp.d Sk` (IEEE mode)
1630	`ln.d Sk` (IEEE mode)
1640	`sin.d Sk` (IEEE mode)
1650	`cos.d Sk` (IEEE mode)

Class 4 subtests

Class 4 subtests, as listed in Table 123, verify the operation of the C3400 Series process control instructions execution in a single CPU environment. For a detailed description of what the process control instructions are, and how they are required to function, refer to the *CONVEX Architecture Reference Manual (C Series)*.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 123
Class 4 subtests

Subtest	Instruction tested
253	`spawn <effa>, Ak`
255	`cfork`
260	`wfork`
263	`join`
265	`idle Sk`

Class 5 Subtests

Class 5 subtests, as listed in Table 124, verify the operation of miscellaneous instructions in a single CPU environment. Specifically, process trapping instructions, loading and storing of the communication registers, and the CPU execution timer synchronization instructions are verified to function correctly in a single CPU environment.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is `0x100`, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table 124
Class 5 subtests

Subtest	Instruction tested
220	`trap #rm, #b`
235	`pbkpt`
240	`ctrsl`
245	`ctrsg`
270	`ldcmr <effa>, Ak`
275	`stcmr <effa>, Ak`
300	Communication register modify bit test

Multiprocessor diagnostics (cpu4333)

16

The cpu4333 test verifies the operation of the CPU complex in a multiheaded environment. Included are tests for concurrent access and use of communication registers, memory, thread creation and termination instructions, interrupts, CPU execution timers, privileged instructions, and exceptions. Correct execution of processors executing in both the same and in different communication index registers (CIRs) is also verified.

Prerequisites and required equipment

An overall view of what part of the system is being tested and which field replaceable units (FRUs) are required for the test to run is illustrated in Table 125.

Table 125
`cpu4333` functional areas tested

Functional area	Tested by the diagnostic	Exercised by the diagnostic
CPU	Yes	No
CUJ	Yes	No
Memory even	No	Yes
Memory odd	No	Yes
PI2	No	Yes
CCU	No	No
SP5	No	Yes

In order to run the `cpu4333` test, the boards listed in Table 126 must be operational. The table shows the tests used to verify the required boards.

Table 126
`cpu4333` required functional boards

Board	Test to verify
Service processor (SP5)	`spu1000, spu4000`
Memory system (MCM, MCM2, MCM3)	`mem4100`
PBUS interface adapter (PI2)	`pi2_4000`
CPU utility board (CUJ)	`CUJ_SCAN`
Central processing unit (CPU)	`cpu4030`

Note

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test invocation

To invoke the cpu4333 test, use the procedure shown in Figure 179.

Figure 179
cpu4333 test invocation sequence

```
(spu)> cd /mnt/test
(spu)> sysreset
(spu)> dshell

CONVEX DIAGNOSTIC SHELL
: test cpu4333 [-c [<class>...]] [-s [<subtest>...]] [+> <filename>]
```

The prompts and responses appear sequentially on the screen, one line at a time, but all prompts and responses are shown in one figure for convenience.

Note

After entering the dshell command, specific dshell parameters may be changed. Refer to Chapter 7, "Diagnostic shell (dshell)" for more information.

Caution

If the system has just been powered up, if `mem4100` was executed with failures, or if `spu4000` was executed, then `initall` must be executed prior to test execution, but only after the SPU EPROM based self-test has passed. Failure to execute `initall` in these circumstances could result in invalid test results.

Entering only

```
test cpu4333
```

executes all cpu4333 subtests sequentially.

Execute one or more specific classes of subtests or one or more individual subtests by using the -c or -s options, respectively. Detailed information for using these options can be found in Chapter 7, "Diagnostic shell (dshell)".

The [+> <filename>] option allows the test results to be appended to *filename*.

Test parameter menu

Once the test is invoked, a test parameter menu prompts for selection of runtime switches. If you answer **y** to the first prompt, the test is run with all default switches, and no other prompts are provided. If you answer **n** to the first prompt, then a series of prompts are presented.

The prompts, their possible answers in brackets [ ], and their default answers in parentheses ( ) are illustrated in Figure 180.

Figure 180
`cpu4333` test parameter menu

```
Test 'cpu4333.t'                              Thu Nov 19 00:00:00 1985

                    ENTER TEST PARAMETERS
       [ ]   Encloses allowed input ranges or values
       ( )   Encloses the default value
       ^     Returns to the previous prompt
       :nn   Returns to the prompt # nn
       :     Returns to the first unsatisfied prompt
       :?    Reviews previous entries

1. Run default switches? [y,n]                                  (y) ->
2. CPUs to test: [01234567]                                (01234567) ->
3. Forced Faulting Enabled? [y,n]                               (n) ->
4. Fault on Instruction Fetches? [y,n]                          (n) ->
5. Sequential Execution? [y,n]                                  (n) ->
6. Timeout Scale Factor Enabled? [1-100]                        (1) ->
7. Dcache Enabled? [y,n]                                        (y) ->
8. Segment of Execution? [0-7]                                  (0) ->
9. Loop Enabled? [y,n]                                          (n) ->
10: Hard Errors Enabled? [y,n]                                  (y) ->
11: Load CPU Code? [y,n]                                        (y) ->
```

The prompts and responses appear sequentially on the screen, one line at a time, but all the prompts and responses are shown in one figure for convenience.

In Figure 180, prompt 2, `[01234567]` represents all available CPUs in the machine under test.

In Figure 180, prompt 4 is only supplied when prompt 3 is answered with **y**.

Prompt explanations

A description of each prompt and its meaning follows:

```
1: Run default switches? [y/n]                (y) ->
```

If you respond with **y** or **RETURN**, no additional test parameter prompts are displayed and testing begins.

However, if you respond with **n**, additional test parameter prompts are displayed, allowing choices other than the default selections. The following prompts are only displayed and answered if the first prompt is answered with **n**.

```
2: CPUs to test: [01234567]                   (01234567) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selections, represented by `01234567`, consist of all available CPUs. The first CPU in the list is the *master* CPU for this test.

```
3: Forced Faulting Enabled? [y,n]             (n) ->
```

If you answer with **y**, normal force faulting occurs on all data references: the system forces a nonresident data exception to occur on every data reference.

If this option is enabled, subtest execution time is increased and the timeout scale factor requires adjustment to prevent the SPU from terminating the test prematurely.

```
4: Fault on Instruction Fetches? [y,n]        (n) ->
```

If you answer with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt appears only if the previous prompt is answered **y**.

```
5: Sequential Execution? [y,n]                (n) ->
```

If you answer with **y**, the sequential bit in the processor status word (PSW) is set to forced sequential execution mode.

```
6: Timeout Scale Factor [1-100]               (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number entered to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
7: Dcache Enabled? [y,n]                    (y) ->
```

The Dcache is normally enabled. However, if it is suspected to be broken, it can be disabled by entering **n** at this prompt.

```
8: Segment of Execution? [0-7]              (0) ->
```

This prompt is only displayed if forced faulting is not enabled. The segment of execution is contained in bits <31..29> of the program counter (PC).

- If **0** is entered, then bits <31..29> of the PC are `000` and the test is run in ring zero.
- If **1** is entered, then bits <31..29> of the PC are `001` and the test is run in ring one.
- If **2** is entered, then bits <31..29> of the PC are `010` and the test is run in ring two.
- If **3** is entered, then bits <31..29> of the PC are `011` and the test is run in ring three.
- If **4, 5, 6,** or **7** is entered, then bits <31..29> of the PC are `100, 101, 110,` or `111`, respectively, and the test is run in ring four.

Refer to the *CONVEX Architecture Reference Manual (C Series)* for more information concerning the meaning of rings in the machine architecture.

```
9: Loop Enabled? [y,n]                      (n) ->
```

If you answer with **y**, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by pressing **CTRL-C**.

```
10: Hard Errors Enabled? [y,n]              (y) ->
```

If this option is enabled and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled, parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
11: Load CPU Code? [y,n]                    (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code is not reloaded, thus saving time.

Test parameter summary

When all prompts have been answered, the screen displays a test parameter summary, as illustrated in Figure 181, which echoes the prompts that have been answered.

Figure 181
cpu4333 test parameter summary

```
                TEST PARAMETER SUMMARY

Run default switches?                  : n
Cpus to test:                          : 01
Parallel Test Execution?               : y
Forced Faulting Enabled?               : y
Sequential Execution?                  : n
Timeout Scale Factor Enabled?          : 1
Dcache Enabled?                        : y
Segment of Execution?                  : 0
Loop Enabled?                          : n
Hard Errors Enabled?                   : y
Load CPU Code?                         : y
```

The actual summary varies depending on the answers to the prompts.

Hardware initialization sequence

After the last prompt is entered, and before test code execution, the following events occur:

- Each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CUJ, PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- Clocks are turned on to all processors selected. If parallel execution is selected, all clocks are turned on. If sequential execution is selected, each CPU's clocks are turned on one at a time, in the order the CPUs are selected.

Note

The first two events are accomplished at the initial start of the `cpu4333` test. The remaining events are accomplished when each subtest is initialized.

Current memory allocation

Immediately before test code execution, a current memory allocation screen is displayed. Figure 182 is an example of the current memory allocation screen.

Figure 182
cpu4333 current memory allocation screen

```
                    Current Memory Allocation

File No.  Physical Address Pid File Name              Logical Offset
      1   00000000-00027fff  0 p0r0_4333             00000000
      2   00028000-000b1fff  0 cpu4333.rnn           00022000
      1   000b2000-000d9fff  1 p0r0_4333             00000000
      2   000da000-00163fff  1 cpu4333.rnn           00022000
          03ff7000-03ff9fff  1 ptet   NA
          03ffa000-03ffafff  1 pte2   NA
          03ffb000-03ffdfff  0 ptet   NA
          03ffe000-03ffefff  0 pte2   NA
          03fffc00-3fffffff  1 pte1   NA
```

The physical and logical addresses shown, as well as the file names, are only representative; the actual addresses will vary depending on installed memory and file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First column**—File number.
- **Second column**—Physical memory addresses where the specified file is loaded (useful in conjunction with the mm(1d) utility).
- **Third column**—Process identification.
- **Fourth column**—File name. (The actual path is */filename* or /mnt/test/CPU/*filename*. The entries `pte1`, `pte2`, and `ptet` are not actual files, but are indications of the page tables.)
- **Fifth column**—Logical starting address of the specified file.

Class descriptions

There are fourteen classes of subtests for `cpu4333`, and they are described in Table 127.

Table 127
cpu4333 class descriptions

Class	Description
1	Communication register single headed tests
2	Communication register multiheaded tests (synchronized stepped, same communication register)
3	Communication register multiheaded tests (synchronized, same communcation register
4	Communication register multiheaded tests (synchronized, different same communication register)
5	Memory instruction single-headed tests
6	Memory instruction multiheaded tests (synchronized stepped, same communication register)
7	Memory instruction multiheaded tests (synchronized, same communication register)
8	Memory instruction multiheaded tests (synchronized, different same communication register)
9	Multiheaded parallel instruction tests
10	Communication register loading and storing tests
11	Multiheaded *trap* and *pbkpt* instruction tests
12	Multiheaded remote invalidates (testing same blocks)
13	Multiheaded remote invalidates (testing different memory blocks)
14	Multiheaded timer, interrupt, and exception processing test

Subtest descriptions

Table 128 lists each `cpu4333` subtest in its order of execution. The table lists each subtest, its class, a description of the subtest, the subtest source file, the minimum time required to run the subtest (nominal time), and the timeout limit (maximum time).

In Table 128, the "Instruction tested/test performed" column lists either a description of what is tested or the instruction that is tested. For more information on an instruction's meaning, refer to the *CONVEX C Series Assembly Language Reference Manual*. The object module (the executable code) for all `cpu4333` subtests is `cpu4333.rnn`.

Table 128
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
101	1	`lck <Ceffa>` (single head)
102	2	`lck <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
103	3	`lck <Ceffa>` (multiple heads, synchronized on same communication register locations)
104	4	`lck <Ceffa>` (multiple heads, synchronized on different communication registers)
111	1	`ulk <Ceffa>` (single head)
112	2	`ulk <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
113	3	`ulk <Ceffa>` (multiple heads, synchronized on same communication register locations)
114	4	`ulk <Ceffa>` (multiple heads, synchronized on different communication registers)
121	1	`tst <Ceffa>` (single head)
122	2	`tst <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
123	3	`tst <Ceffa>` (multiple heads, synchronized on same communication register locations)
124	4	`tst <Ceffa>` (multiple heads, synchronized on different communication registers)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
131	1	`get.w <Ceffa>, Ak` (single head)
132	2	`get.w <Ceffa>, Ak` (multiple heads, synchronized, stepped on same communication registers)
133	3	`get.w <Ceffa>, Ak` (multiple heads, synchronized on same communication register locations)
134	4	`get.w <Ceffa>, Ak` (multiple heads, synchronized on different communication registers)
141	1	`get.l <Ceffa>, Sk` (single head)
142	2	`get.l <Ceffa>, Sk` (multiple heads, synchronized, stepped on same communication registers)
143	3	`get.l <Ceffa>, Sk` (multiple heads, synchronized on same communication register locations)
144	4	`get.l <Ceffa>, Sk` (multiple heads, synchronized on different communication registers)
151	1	`put.w Ak, <Ceffa>` (single head)
152	2	`put.w Ak, <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
153	3	`put.w Ak, <Ceffa>` (multiple heads, synchronized on same communication register locations)
154	4	`put.w Ak, <Ceffa>` (multiple heads, synchronized on different communication registers)
161	1	`put.l Sk, <Ceffa>` (single head)
162	2	`put.l Sk, <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
163	3	`put.l Sk, <Ceffa>` (multiple heads, synchronized on same communication register locations)
164	4	`put.l Sk, <Ceffa>` (multiple heads, synchronized on different communication registers)
171	1	`rcv.w <Ceffa>, Ak` (single head)
172	2	`rcv.w <Ceffa>, Ak` (multiple heads, synchronized, stepped on same communication registers)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
173	3	`rcv.w <Ceffa>, Ak` (multiple heads, synchronized on same communication register locations)
174	4	`rcv.w <Ceffa>, Ak` (multiple heads, synchronized on different communication registers)
181	1	`rcv.l <Ceffa>, Sk` (single head)
182	2	`rcv.l <Ceffa>, Sk` (multiple heads, synchronized, stepped on same communication registers)
183	3	`rcv.l <Ceffa>, Sk` (multiple heads, synchronized on same communication register locations)
184	4	`rcv.l <Ceffa>, Sk` (multiple heads, synchronized on different communication registers)
191	1	`snd.w Ak, <Ceffa>` (single head)
192	2	`snd.w Ak, <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
193	3	`snd.w Ak, <Ceffa>` (multiple heads, synchronized on same communication register locations)
194	4	`snd.w Ak, <Ceffa>` (multiple heads, synchronized on different communication registers)
201	1	`snd.l Sk, <Ceffa>` (single head)
202	2	`snd.l Sk, <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
203	3	`snd.l Sk, <Ceffa>` (multiple heads, synchronized on same communication register locations)
204	4	`snd.l Sk, <Ceffa>` (multiple heads, synchronized on different communication registers)
211	1	`inc.w <Ceffa>, Ak` (single head)
212	2	`inc.w <Ceffa>, Ak` (multiple heads, synchronized, stepped on same communication registers)
213	3	`inc.w <Ceffa>, Ak` (multiple heads, synchronized on same communication register locations)
214	4	`inc.w <Ceffa>, Ak` (multiple heads, synchronized on different communication registers)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
221	1	`inc.l <Ceffa>, Sk` (single head)
222	2	`inc.l <Ceffa>, Sk` (multiple heads, synchronized, stepped on same communication registers)
223	3	`inc.l <Ceffa>, Sk` (multiple heads, synchronized on same communication register locations)
224	4	`inc.l <Ceffa>, Sk` (multiple heads, synchronized on different communication registers)
231	1	`mat.w Ak, <Ceffa>` (single head)
232	2	`mat.w Ak, <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
233	3	`mat.w Ak, <Ceffa>` (multiple heads, synchronized on same communication register locations)
234	4	`mat.w Ak, <Ceffa>` (multiple heads, synchronized on different communication registers)
241	1	`mat.l Sk, <Ceffa>` (single head)
242	2	`mat.l Sk, <Ceffa>` (multiple heads, synchronized, stepped on same communication registers)
243	3	`mat.l Sk, <Ceffa>` (multiple heads, synchronized on same communication register locations)
244	4	`mat.l Sk, <Ceffa>` (multiple heads, synchronized on different communication registers)
251	5	`sndr.w Ak, <effa>` (single head)
252	6	`sndr.w Ak, <effa>` (multiple heads, synchronized, stepped on same resource)
253	7	`sndr.w Ak, <effa>` (multiple heads, synchronized on same resources)
254	8	`sndr.w Ak, <effa>` (multiple heads, synchronized on different resources)
261	5	`sndr.l Sk, <effa>` (single head)
262	6	`sndr.l Sk, <effa>` (multiple heads, synchronized, stepped on same resource)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
263	7	`sndr.l Sk, <effa>` (multiple heads, synchronized on same resources)
264	8	`sndr.l Sk, <effa>` (multiple heads, synchronized on different resources)
271	5	`rcvr.w <effa>, Ak` (single head)
272	6	`rcvr.w <effa>, Ak` (multiple heads, synchronized, stepped on same resource)
273	7	`rcvr.w <effa>, Ak` (multiple heads, synchronized on same resources)
274	8	`rcvr.w <effa>, Ak` (multiple heads, synchronized on different resources)
281	5	`rcvr.l <effa>, Sk` (single head)
282	6	`rcvr.l <effa>, Sk` (multiple heads, synchronized, stepped on same resource)
283	7	`rcvr.l <effa>, Sk` (multiple heads, synchronized on same resources)
284	8	`rcvr.l <effa>, Sk` (multiple heads, synchronized on different resources)
311	5	`pshr Ak, <effa>` (single head)
312	6	`pshr Ak, <effa>` (multiple heads, synchronized, stepped on same resource)
313	7	`pshr Ak, <effa>` (multiple heads, synchronized on same resources)
314	8	`pshr Ak, <effa>` (multiple heads, synchronized on different resources)
321	5	`popr <effa>, Sk` (single head)
322	6	`popr <effa>, Sk` (multiple heads, synchronized, stepped on same resource)
323	7	`popr <effa>, Sk` (multiple heads, synchronized on same resources)
324	8	`popr <effa>, Sk` (multiple heads, synchronized on different resources)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
331	5	`incr.w <effa>, Ak` (single head)
332	6	`incr.w <effa>, Ak` (multiple heads, synchronized, stepped on same resource)
333	7	`incr.w <effa>, Ak` (multiple heads, synchronized on same resources)
334	8	`incr.w <effa>, Ak` (multiple heads, synchronized on different resources)
341	5	`incr.l <effa>, Sk` (single head)
342	6	`incr.l <effa>, Sk` (multiple heads, synchronized, stepped on same resource)
343	7	`incr.l <effa>, Sk` (multiple heads, synchronized on same resources)
344	8	`incr.l <effa>, Sk` (multiple heads, synchronized on different resources)
351	9	`pfork <effa>, Ak` (multiple headed pfork with an outstanding fork request)
352	9	`pfork <effa>, Ak` (multiple headed pfork with no outstanding fork request)
353	9	`pfork <effa>, Ak` (multiple headed pfork with no outstanding fork request)
354	9	`pfork <effa>, Ak` (verify all threads can be uniquely picked up)
361	9	`cfork` (multiple headed cfork with no outstanding fork request)
362	9	`cfork` (multiple headed cfork with an outstanding fork request)
371	9	`wfork` (multiple headed wfork)
372	9	`wfork` (verify a wfork spins if thread register is unlocked)
381	9	`spawn <effa>, Ak` (multiple headed spawn with an outstanding fork request)
382	9	`spawn <effa>, Ak` (multiple headed spawn with no outstanding fork request)
383	9	`spawn <effa>, Ak` (multiple headed spawn with no outstanding fork request)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
384	9	`spawn <effa>, Ak` (verify all threads can be uniquely picked up)
391	9	`join` (multiple headed wfork)
392	9	`join` (verify a join spins if thread register is unlocked)
401	9	`idle` (multiple headed idle)
431	5	`tas <effa>` (single head)
432	6	`tas <effa>` (multiple heads, synchronized, stepped on same memory)
433	7	`tas <effa>` (multiple heads, synchronized on same memory locations)
434	8	`tas <effa>` (multiple heads, synchronized on different memory)
441	5	`tac <effa>` (single head)
442	6	`tac <effa>` (multiple heads, synchronized, stepped on same memory)
443	7	`tac <effa>` (multiple heads, synchronized on same memory locations)
444	8	`tac <effa>` (multiple heads, synchronized on different memory)
451	10	`ldcommunication register <effa>, Ak` (single head)
452	10	`ldcommunication register <effa>, Ak` (multiple heads, same communication register set)
453	10	`ldcommunication register <effa>, Ak` (multiple heads, different communication register set)
461	10	`stcommunication register Ak, <effa>` (single head)
462	10	`stcommunication register Ak, <effa>` (multiple heads, different communication register set)
471	11	`trap #rm, #bit` (multiple heads, single head execution, same CIR)
472	11	`trap #rm, #bit` (multiple heads, single head execution, different CIR)
476	11	`pbkpt` (multiple heads, single head execution, same CIR)
477	11	`pbkpt` (multiple heads, single head execution, different CIR)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
500	12	Remote invalidate 1 (single head writing bytes, other heads spin on communication register)
501	12	Remote invalidate 1 (single head writing halfwords, other heads spin on communication register)
502	12	Remote invalidate 1 (single head writing words, other heads spin on communication register)
503	12	Remote invalidate 1 (single head writing longwords, other heads spin on communication register)
510	12	Remote invalidate 2 (single head writing bytes, other heads spin on memory)
511	12	Remote invalidate 2 (single head writing halfwords, other heads spin on memory)
512	12	Remote invalidate 2 (single head writing words, other heads spin on memory)
513	12	Remote invalidate 2 (single head writing longwords, other heads spin on memory)
520	12	Remote invalidate 3 (single head writing bytes, other heads delayed reading)
521	12	Remote invalidate 3 (single head writing halfwords, other heads delayed reading)
522	12	Remote invalidate 3 (single head writing words, other heads delayed reading)
523	12	Remote invalidate 3 (single head writing longwords, other heads delayed reading)
530	12	Remote invalidate 4 (multiple heads writing unique bytes in 4 kbyte block)
531	12	Remote invalidate 4 (multiple heads writing unique halfwords in 4 kbyte block)
532	12	Remote invalidate 4 (multiple heads writing unique words in 4 kbyte block)
533	12	Remote invalidate 4 (multiple heads writing unique longwords in 4 kbyte block)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
540	13	Remote invalidate 1 (single head writing bytes, other heads spin on communication register)
541	13	Remote invalidate 1 (single head writing halfwords, other heads spin on communication register)
542	13	Remote invalidate 1 (single head writing words, other heads spin on communication register)
543	13	Remote invalidate 1 (single head writing longwords, other heads spin on communication register)
550	13	Remote invalidate 1 (single head writing bytes, other heads spin on communication register)
551	13	Remote invalidate 1 (single head writing halfwords, other heads spin on communication register)
552	13	Remote invalidate 1 (single head writing words, other heads spin on communication register)
553	13	Remote invalidate 1 (single head writing longwords, other heads spin on communication register)
560	13	Remote invalidate 1 (single head writing bytes, other heads spin on communication register)
561	13	Remote invalidate 1 (single head writing halfwords, other heads spin on communication register)
562	13	Remote invalidate 1 (single head writing words, other heads spin on communication register)
563	13	Remote invalidate 1 (single head writing longwords, other heads spin on communication register)
570	13	Remote invalidates with PTE missing
571	13	Remote invalidates with read fault
611	5	`putr.w Ak, <effa>`, single head
612	6	`putr.w Ak, <effa>`, multiple heads (synchronized, stepped on same resource)
613	7	`putr.w Ak, <effa>`, multiple heads (synchronized on same resources)

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
614	8	`putr.w Ak, <effa>`, multiple heads (synchronized on different resources)
621	5	`putr.l Sk, <effa>`, single head
622	6	`putr.l Sk, <effa>`, multiple heads (synchronized, stepped on same resource)
623	7	`putr.l Sk, <effa>`, multiple heads (synchronized on same resources)
624	8	`putr.l Sk, <effa>`, multiple heads (synchronized on different resources)
631	5	`getr.w Ak, <effa>`, single head
632	6	`getr.w Ak, <effa>`, multiple heads (synchronized, stepped on same resource)
633	7	`getr.w Ak, <effa>`, multiple heads (synchronized on same resources)
634	8	`getr.w Ak, <effa>`, multiple heads (synchronized on different resources)
641	5	`getr.l Sk, <effa>`, single head
642	6	`getr.l Sk, <effa>`, multiple heads (synchronized, stepped on same resource)
643	7	`getr.l Sk, <effa>`, multiple heads (synchronized on same resources)
644	8	`getr.l Sk, <effa>`, multiple heads (synchronized on different resources)
651	5	`matr.w Ak, <effa>`, single head
652	6	`matr.w Ak, <effa>`, multiple heads (synchronized, stepped on same resource)
653	7	`matr.w Ak, <effa>`, multiple heads (synchronized on same resources)
654	8	`matr.w Ak, <effa>`, multiple heads (synchronized on different resources)
661	5	`matr.l Sk, <effa>`, single head

Table 128 (continued)
cpu4333 subtests

Subtest	Class	Instruction tested/test performed
662	6	`matr.l Sk, <effa>`, multiple heads (synchronized, stepped on same resource)
663	7	`matr.l Sk, <effa>`, multiple heads (synchronized on same resources)
664	8	`matr.l Sk, <effa>`, multiple heads (synchronized on different resources)
671	5	`casr`, single head
1000	14	Interrupt test, multiple heads
1010	14	Deadlock test, multiple heads
1020	14	`patu test`, multiple heads
1025	14	`pate test`, multiple heads
1030	14	`ctrsl test`, multiple heads
1035	14	`ctrsg test`, multiple heads
1040	14	Timer ring cross test, multiple heads
1045	14	CPU execution timer test, multiple heads

Index

Symbols

A

B

C

D

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Order Number
DHW-302

Document Number
760-003830-000