CONVEX Diagnostic Utilities Manual (C1, C120)

Document No. 760-001050-201

Second Edition
October 1988

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX Diagnostic Utilities Manual
(C1, C120)
Order No. DHW-072
Second Edition

Printed in the United States of America

Revision Sheet

CONVEX Diagnostic Utilities Manual
(C1, C120)

Edition	Document No.	Date	Description
First	760-001050-200	April 1988	Contains portions of material previously found in the *CONVEX SPU UNIX Programmer's Manual.*
Second	760-001050-201	October 1988	Added six online manual pages (*errintd.1d, get_defects.1d, hard_logger.1d, mm_sniff.1d, security_clear.1d, softlog.5d*), updated one online manual page (*DB_diskfmt.5d*), and updated the permuted index and table of contents.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

Appendixes

List of Tables

List of Figures

Alphabetical Listing of Man Pages

1D. Diagnostic Commands

5D. Diagnostic File Formats

THIS PAGE INTENTIONALLY LEFT BLANK

Permuted Index – Utilities

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and Intended Audience

This document presents the diagnostic utilities for CONVEX C1 and C120 computers. The material presented describes the features of the Service Processor Unit (SPU) operating system and contains all diagnostic utilities for the CONVEX C1 and C120 machines.

The *CONVEX Diagnostic Utilities Manual (C1, C120)* is a reference tool for CONVEX personnel who use the diagnostic utilities, CONVEX customers who perform their own maintenance, and the CONVEX diagnostics sustaining staff.

Scope

This manual applies to CONVEX C1 and C120 computers.

Outline

This manual contains information on the SPU UNIX environment, a detailed explanation of the Diagnostics Shell (Dshell) and the scan utilities, diagnostic utilities, and diagnostic file formats. This manual is divided into the following chapters:

Chapter 1. Diagnostic Environment—Introduces and explains the diagnostic and SPU UNIX operating system environments as well as describes the directory and file structure

Chapter 2. Diagnostic Shell (Dshell)—Introduces and explains the Dshell

Chapter 3. Scan Utility—Provides a detailed introduction and explanation of scan utility

Chapter 4. Diagnostic Utilities—Describes publicly-accessible diagnostic commands in alphabetical order

Chapter 5. Diagnostic File Formats—Contains detailed explanations of all pertinent diagnostic file formats

Appendix A. Problem Reporting—Contains information concerning how to use the *contact* facility to report problems

Notational Conventions

All entries are based on a common format, not all of which will always appear:

- The NAME subsection lists the exact names of the commands and subroutines covered under the entry and gives a short description of their purpose.
- The SYNOPSIS summarizes the use of the program being described.
- The DESCRIPTION subsection discusses in detail the subject at hand.
- The FILES subsection gives the names of files that are built into the program.
- A SEE ALSO subsection gives pointers to related information. Each cross-referenced command is boldfaced and contains the filename plus a number in parentheses. The number is parentheses is the chapter number for the cross-reference. Any cross-reference with a (2), (3), (4), (6), or (7) will not be found in this manual.
- A DIAGNOSTICS subsection discusses the diagnostic indications that may be produced. Messages that are intended to be self-explanatory are not listed.
- The BUGS subsection gives known bugs and sometimes deficiencies. Occasionally the suggested fix is described.

A few conventions are used, particularly in the "Commands" subsection:

- **Boldface** words are considered literals, and are typed just as they appear.
- Square brackets ([ ]) around an argument indicate that the argument is optional. When an argument is given as "name," it always refers to a filename.
- Ellipses (...) are used to show that the previous argument-prototype may be repeated.
- All CONVEX illustrations have an illustration catalog number at the bottom right-hand corner that is for CONVEX use only.

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

WARNING

Warnings highlight procedures or information necessary to avoid injury to personnel. Warnings immediately precede the critical information and include a description of the hazard.

CAUTION

Cautions highlight procedures or information necessary to avoid damage to equipment, damage to software, or loss of data. Cautions immediately precede the critical information and include a description of the possible damage.

> **NOTE**
>
> Notes highlight useful information that is supplemental in nature. Notes may immediately precede or follow the information that is being highlighted.

Associated Documents

Readers should become familiar with both the glossary of technical terms in Appendix A. A feedback form is found in the rear of the handbook; readers are invited to comment on the service and clarity of this manual.

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

- *CONVEX Processor Diagnostics Manual (C1, C120)*, Order No. DHW-071
- *CONVEX PBUS I/O System Diagnostics Manual*, Order No. DHW-008
- *CONVEX Processor Operation Guide (C100 Series, C200 Series)*, Order No. DHW-015

Ordering Documentation

To obtain the most current version of this or any other CONVEX document, order using the 6-digit order number. If the order number is not known, order by the exact title. In some situations the most current version is not desired. In order to receive a specific version of a manual, the manual must be ordered by a 12-digit part, or document, number, which can be provided by CONVEX.

This order number for this manual is DHW-072
The document number for this manual is 760-001050-201.

CONVEX documents can be ordered by mail by sending a request to:

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

Technical Assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC). The TAC can be reached in Texas by calling (214)952-0379, or by calling 1(800)952-0379 from other locations in the continental United States. Customers outside the United States should contact their local CONVEX office.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1
Diagnostics Environment

1.1 Overview

CONVEX system diagnostics consist of a suite of test programs designed (except where noted) to execute under the Service Processor operating system, SPU UNIX. These programs use the capabilities of the Service Processor to test the operation of one or more of the functions of the system and to report any errors detected. All diagnostics in this manual are intended to be executed off-line, that is, while CONVEX UNIX is not being executed by any of the Central Processing Units (CPUs) in the system.

The Service Processor, together with SPU UNIX and the various diagnostic utilities and test programs, comprise the CONVEX diagnostic environment. For more information about the diagnostic environment, refer to the "Diagnostic Environment" chapter in the *CONVEX Processor Diagnostics Manual (C200 Series)* or the *CONVEX Processor Diagnostics Manual (C1, C120)*, depending on the architecture of the machine under test. This chapter describes the hardware and software components of this environment and is intended to provide the background necessary to fully utilize the capabilities of the CONVEX processor diagnostics.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2
Diagnostic Shell (Dshell)

2.1 Introduction

The Diagnostic Shell (Dshell) is a command interface program that runs on the Service Processor Unit (SPU). Most diagnostics are interfaced through the Dshell. 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 controlling data logging and test repetition.

This chapter is particularly helpful to field service and manufacturing personnel who are responsible for board or system verification, or board or system debugging.

2.2 Dshell Overview

The Dshell has two basic functions:

- Selecting diagnostics for execution
- Selecting 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

2.3 Dshell Commands for Manual Mode

This section explains the commands available under the Dshell.

To access the Dshell, enter **dshell<CR>** at the spu> prompt. All Dshell commands can be entered at the Dshell prompt (:).

2.3.1 *access*

The *access* command (*!*) is used to access, or *fork*, a UNIX shell to execute the command that follows *!*. Any single command available under SPU UNIX can follow the *!*. Once the command

has executed, program control reverts to the Dshell. The standard form of this command is:

! [unix_command]

2.3.2 *exit*

There are four different forms of exits available in the Dshell. They exit from tests back to the Dshell command level and from the Dshell to the UNIX command level. The following table shows the standard forms of these commands:

Table 2–1, *exit* Commands

OPTION	FUNCTION
exit (or *e*) or *quit* (or *q*)	Immediate termination of the Dshell process and any test processes that may have been forked

^C	Returns user to the Dshell command level if no subtest is running. If subtest execution has started, the following test menu appears:
	`0 continue test`
	`1 abort current test`
	`2 abort current subtest`
	`3 abort and pause at end of current subtest`

^B	Immediately terminate the Dshell and any associated active processes. Core is dumped.

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 *^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 *^C* commands allow this clean-up routine to execute.

The *^B* command is a dirty, nonmaskable exit that pays no heed to what is currently being executed. Sensitive code is not protected from *^B*. In addition, *^B* 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 *^B* only as a last resort. System initialization should immediately follow *^B* execution.

Using *^C* after a test command has been entered, but before the SPU has successfully forked the test process, may not kill the test process.

2.3.3 *help*

This command causes a standard *help* menu to be displayed. The menu describes the correct command syntax for each Dshell command and gives a terse description of what each command does. The standard form of this command is:

help

The *-h* option is available on all other commands to display that command's help menu. The standard form of this command is:

[*command*] *-h*

2.3.4 *status*

The *status* command generates a report on the current state of the Dshell command options. This report gives the name of each flag, its current value, and an explanation of its current effect. The standard form of the *status* command is:

status

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 *^B*.

2.4 Manual Mode of Execution

In the manual mode, the sequencing and parameters for diagnostic testing can be determined either directly or indirectly via Dshell. Manual execution allows selection of the conditions under which subtests are executed. Using this mode of operation returns low-level pass or fail data.

2.4.1 Dshell Commands for Manual Mode

This section explains the commands available under the Dshell for the manual mode of execution.

2.4.1.1 *log*

Normally, Dshell diagnostics terminate after a single failure. The *log* command provides a mechanism for specifying the number of failures that will be allowed to occur before a test or subtest terminates execution.

The standard form of the *log* command is:

log [*options*]

The options available include invoking the command at both the subtest (*-s*) and test (*-t*) levels, as well as selecting the number of failures to be allowed. The default setting is *log off -s -t*.

The following table lists *log* options:

Table 2–2, *log* Options

OPTION	FUNCTION
log off	Disable all multiple failure logging
log off [*-s*] [*-t*]	Disable previous log entries at the subtest or test level and terminating test after first failure
log -s [*nn*]	Allows the tests to run until *nn* subtest failures have been logged
log -t [*nn*]	Allows subtests to run until *nn* failures have been logged

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 enabled.

2.4.1.2 *loop*

The *loop* command causes the Dshell to repeat the execution of a test or subtest, as indicated in the following table:

Table 2–3, *loop* Options

OPTION	FUNCTION
loop off	Disables all looping
loop off [*-s*] [*-t*]	Disables subtest or test looping
loop -s	Loops on every subtest
loop -s [*nn*]	Loops on subtest *nn* if it is executed
loop -t	Loops on entire test

Subtests or tests continue executing until the loop mode is disabled. The standard form of the command is:

loop [*options*]

As with the *log* command, options include *-s* and *-t*, which enables the command to execute at either the subtest or test level, and the subtest number, if looping on a specific subtest is desired. The default setting is *loop off -s -t*.

To continue test execution after looping on a subtest, enter a *pause* command before test execution. This allows for disabling subtest looping when desired (*loop off -s*) and continuing normal test execution.

As an added feature, if test looping is enabled, the Dshell records the pass or fail result of each test iteration. When *^C* is executed and test looping terminates, the results are summarized and written to the selected output.

2.4.1.3 *msgs*

The *msgs* command enables or disables different levels of test, class, and subtest result messages. The following table lists the options available for all tests:

Table 2–4, *msgs* Options

OPTION	PURPOSE
msgs off [*-f*] [*-s*] [*-t*]	Disables all or specific test messages
msgs -f [*long*] or [*short*]	Enables long or short failure messages
msgs -s	Enables subtest result messages
msgs -t	Enables test result messages

The basic format of the *msgs* command is:

msgs [*options*]

The default setting is *msgs -f long -s -t.*

2.4.1.4 *pause*

The *pause* command 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

The following table shows how the available options may be used singularly or in combination:

Table 2–5, *pause* Options

OPTION	FUNCTION
pause off	Disables all pauses
pause off [*-f*] [*-b*] [*-e*]	Disables specified pauses
pause -f	A pause if a failure is encountered during execution of any subtest
pause -f [*nn*]	A pause if a failure is encountered during execution of subtest *nn*
pause -b	A pause at the beginning of each subtest that is executed
pause -b [*nn*]	A pause at the beginning of subtest *nn* if it is executed
pause -e	A pause at the end of each subtest that is executed
pause -e [*nn*]	A pause at the end of subtest *nn* if it is executed

The standard form of the *pause* command is:

pause [*option*] [*nn*]

where *nn* signifies the number of a specific subtest.

The default setting is *pause off*.

Omitting the argument *nn* causes a pause to be executed for each subtest (where applicable).

Each of the variations of *pause* can be set up either for all subtests or for a particular subtest.

When a pause occurs, program control is returned to the Dshell command level. Any UNIX command or diagnostic utility can then be executed by using the *access* command (*!*). When failure analysis or UNIX command execution has been completed, entering <CR> causes the test sequence to be re-engaged.

> NOTE
>
> The test has no knowledge of access commands occurring. If the machine state is destroyed, 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 multiple failures per test (*log -t*) are not enabled, or when the current failure is the last failure permitted by the count entered with a *log -t* command.

2.4.1.5 *test*

The *test* command executes specific tests, and displays test, class, and subtest menus, as indicated in the following table:

Table 2–6, *test* Options

OPTION	FUNCTION
test	Presents a menu listing all tests available under the Dshell
test [*testname*]	Executes all subtests for the test named
test [*testname*] *-c*	Presents the class menu for the test named; user prompted to select class(es) for execution
test [*testname*] *-s*	Presents the subtest menu for the test named; user prompted to select subtest(s) for execution
test [*testname*] *-s* [*subtest list*]	Executes subtests in the order specified
test [*testname*] *-c* [*class list*]	Executes subtests (within the classes listed) in the order specified

To access a menu only, enter

test<CR>

at the Dshell prompt (:). This command displays a menu of all test available within the current working directory, as well as the tests located in */mnt/test*. The following menu is displayed:

Figure 2–1, 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.
```

At the menu, a specific test can be executed or additional menus displayed. For example, for a specific subtest menu, answer the prompt with

test [testname] -s<CR>

where *testname* is the name of the test to be executed (refer to Table 2–6, *test* Options). The string *-s* is a flag that causes the subtest menu to be displayed. Similarly, the string *-c* causes the class menu to be displayed. The previous menu commands function in the class and subtest menu levels as well.

The standard form of the *test* command for test execution is:

test [*testname*] [*options*]

where *testname* refers to the name of the test to be executed. The *options* (*-s* and *-c*) enable various operations with test classes and subtests to occur.

To redirect input and output, use the following notational conventions:

- < — Causes input to be taken from the file listed immediately after the symbol
- > — Causes standard output to be directed to the file (but not the screen) listed immediately after the symbol
- +> — Causes standard output to be appended to both the following file and to the screen

To use the options for specific Dshell commands (*log, loop, pause,* etc.), enter those commands before entering the specific test to execute (or enter the name of the script file to execute). For example, to run *dev4400* subtests until three failures have been logged with *pause off*, enter

log -s 3<CR>
pause off<CR>
test dev4400<CR>

This causes *dev4400* to begin execution and run until three failures occurred or until test completion.

Once the colon prompt is displayed, any new Dshell command can be initiated.

< filename, > filename, and *+> filename* may be appended to the end of any of form of the *test* command. However, > and +> are mutually exclusive.

Default test execution is defined to be execution of each subtest that is available in a test, once, in ascending order.

When entering a test command of the type shown in the last two cases in Table 2–6, *test* Options, it is possible to arrange the execution order of the subtests or classes specified. It is also possible to execute multiple iterations of specific subtests or classes, or groups of subtests or classes. This is best illustrated by the examples illustrated in the following table:

Table 2–7, Subtest Configurations

OPTION	FUNCTION
-s 1	Subtest 1
-s 1,5	Subtests 1,5
-s 1-5	Subtests 1,2,3,4,5
-s 2(1-5)	Subtests 1,2,3,4,5,1,2,3,4,5
-s 1,5,3(5,4)	Subtests 1,5,5,4,5,4,5,4

> **NOTE**
>
> Subtest and class specification syntax is identical.

2.4.2 Dshell Script Files

The following listing typifies a Dshell script file that will execute Convex CPU Instruction Set diagnostics. For the purposes of this example, assume the name of the file is *D_novec*:

```
pause off
pause -f
log off
log -t 999
msgs -f long
msgs -t
msgs -s
test cpu4000 -s 10-499,800-919 <T_ring0 +>cpu4000.0
test cpu4000 -s 10-499,800-919 <T_fault_icache +>cpu4000.0
```

Several points should be noted:

1. The *pause off* command disables all pauses that were enabled before script execution.
2. 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.
3. The *log off* command disables previously enabled multiple failure logging.
4. The *log -t 999* command instructs the Dshell to allow multiple subtest failures per test without terminating the test.
5. The *msgs* commands set up long-failure, test, and subtest messages, respectively.
6. 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 *cpu4000.0*. The +> option appends results to the end of the file, leaving previous contents of the file undisturbed.

If this command file is present in */mnt/test* on the SPU, simply enter the Dshell from SPU UNIX, and enter

<D_novec<CR>

This entry invokes the execution of all the commands contained in *D_novec*.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3
Scan Utility

3.1 Introduction

The *scan* utility is an interactive software tool that allows you to control and observe the internal states of individual boards of the CONVEX central processing unit (CPU) and channel control units (CCU's). It provides a means of monitoring the state of the CPU at any time. Thus, you can use *scan* as a debugging tool.

The *scan* utility allows you to define mnemonics (symbolic names) for bit fields within the scan rings and then make a reference to scan ring data using these mnemonics. You can define symbolic synonyms for values within these fields, e.g., on, off, reading, writing, etc. These synonyms can be used in place of their numeric equivalents. Mnemonic definitions and scan ring structure definitions are expressed in a text file called the scan definitions file. A separate compiler is provided to parse and translate the textual description into a binary file. You can create and modify the definitions file with any text editor. The *scan* utility works interactively through either a line-oriented or screen-oriented device.

In addition, the *scan* utility contains a Cshell-type script (control flow language), which operates under control of SPU UNIX. The scan control flow language allows you to develop small files that have elementary conditional flow control to solve some of the more complex problems encountered on the scan rings. This allows you to use the *scan* utility more in a testing mode than as a pattern generator. With a scan script, you can test conditions on the scan ring for correctness and allow different processing to occur once the condition is detected.

Essentially, to use *scan* on a board, you would:

1. Create the definitions file (analogous to declaring variables in a program), *example.scd.*
2. Run the scan compiler (type **scanc example**), which reads the *example.scd* file and creates the file *example.sco.*
3. Copy the *example.sco* file to SPU.
4. Create a script file *sample* (analogous to the executable part of a program) on the SPU, for example, that reads two fields from the board's scan ring, compares the fields, and prints them out if they differ.
5. On the SPU type **scan example**. This runs the *scan* program, which reads *example.sco* (the scan ring definitions).
6. Then at the *scan:* prompt, type **sample**; *scan* then executes the script *sample.*

3.2 Scan Definitions File

The scan definitions file is a text file, which you can create and modify easily with any text editor. It must be compiled with the scan compiler (see "Scan Compiler") before you can use the interactive *scan* utility. The file contains four types of specifications:

- Synonym list specifications (synlist)
- Format specifications (format)
- Ring specifications (ring)
- Screen specifications (screen)

Together these specifications identify the following:

- Bit locations of fields within scan rings.
- Symbolic names of fields
- Synonyms for field values
- Screen formats for interactive edit

You can express the definitions in a simple, block structured syntax, which looks similar to "struct" definitions in the C programming language. You cannot use a forward reference since the compiler is single passed. This presents no problem, however, as long as you ensure that the specifications occur in the following order:

1. All synonym list specifications.
2. All format specifications.
3. All ring specifications.
4. All screen specifications.

3.2.1 Syntax

You can use a free-format syntax in the definitions file; there are no indentation rules or dependencies on end of line. The end of line (newline) is treated as white space, along with tab and blank. You can place comments anywhere white space is allowed. You must enclose the comments within /* and */(just as you do for C). You are not allowed to put a space between / and *; if you do, a compiler error occurs. However, white space may occur anywhere between tokens. For example:

```
/*This is a comment*/

/*
 * This is also a comment ...
*/

/ * Space (like this) causes a compiler error * /
```

Your user-defined symbols must begin with a nondigit and contain only contiguous characters. You are not allowed to use the following characters:

<	left angle bracket
>	right angle bracket
{	left curly bracket
}	right curly bracket
:	colon
;	semicolon
=	equal
/	slash

white space (blank, newline, tab, etc.)

The occurrence of any of these characters terminates the symbol and possibly begins the next token. For instance, a symbol followed by a semicolon would be valid, with the semicolon serving as a symbol termination indicator.

The following examples illustrate valid symbols:

```
cache_state
x_bus_source
wrd_code(1-0)*     /*This is a single valid symbol*/
```

Following are examples of invalid symbols:

```
cash state         /*Actually two symbols*/
source/dest        /*Slash not allowed*/
wrd_code(1..0)*    /*Dot not allowed*/
```

Use any contiguous string of decimal digits, optionally preceded by "-" to represent a numeric value.

3.2.2 Synonym List Specification

Use the synonym list to define synonyms for numeric values. The synonym list specification has the form:

```
synlist SYNAME
  {
      SYMBOL = SCALD_NUMBER;
        .
        .
        .
  };
```

As the example illustrates, begin the synonym list specification with the word *synlist*, followed by the symbolic name (SYNAME); then specify the symbol name followed by the numeric value (SCALD_NUMBER), each followed by a semicolon.

The SCALD_number can be a simple decimal number, or a number with a radix specification of 2 to 16 (expressed in decimal) followed by #, followed by the number in the specified radix. Use the letters a-f to represent digit values of10-15. The following are valid SCALD numbers:

```
36          /* decimal 36 */
2#100010    /* binary 100 010 */
16#ffff     /* hexadecimal ffff */
```

The following example shows how to define a synonym list:

```
synlist fp_staus
    {
            normal = 0;
            overflow = 2#001;
            underflow = 2#010;
            divzero = 2#100;
    };
```

The symbols you use within each synonym list must be unique with respect to other symbols in the **same** list; however, you are allowed to use them again in other synonym lists, using different values. Thus, this allows you to use the name of a synonym list to establish a context for the interpretation of field values. For example, the following two synonym lists can be used in the specification of fields where 1 or 0 indicates an *on* condition:

```
synlist on_off
    {
            on = 1;
            off = 0;
    }; /* END synlist on_off */

synlist on_off* /* inverted on/off logic */
    {
            on = 0;
            off = 1;
    }; /* END synlist on_off* */
```

You can use the synonym list *on_off* in the specification of fields where 1 indicates an *on* condition; whereas, the *on_off** synonym list can be used for fields that interpret 0 to be *on*. This capability relieves you from remembering where logic is inverted.

3.2.3 Format Specification

The ring format specification (or format specification) defines fields within rings. When you develop the format specification, you must include the symbolic name of the field and the location of the bits composing the field. Including the associated synonym list is optional. The format specification has the form:

```
format FORNAME
    {
            FIELD_DEFINITION;
            . /* as many as you want */
            .
            .
    };
```

As the example illustrates, begin the format specification with the word *format*, followed by a symbolic name (FORNAME) for the field's name, followed by one or more field definitions (FIELD_DEFINITION), each terminated by a semicolon.

The FIELD_DEFINITION begins with a symbolic name that represents the name of the field, optionally followed by a BIT_FIELD_DESCRIPTOR, optionally followed by a colon, and synonym listname, and terminated with a semicolon. Thus, it has the form:

FIELDNAME [BIT_FIELD_DESCRIPTOR] [:SYNAME];

The BIT_FIELD_DESCRIPTOR indicates the bits in the scan ring that comprise the field. The bits need not be contiguous. The format for the BIT_FIELD_DESCRIPTOR is general enough to allow a contiguous range of bits, with either end being most significant, or a random list of bits. The BIT_FIELD_DESCRIPTOR may be any of the following forms:

< BIT >
< BIT .. BIT >
< BIT .. BIT : STEP >
< BIT : WIDTH >
<: WIDTH > ******extension*****
< BIT : WIDTH : STEP >
< : WIDTH : STEP > ******extension******
< BIT_LIST >

BIT and STEP are nonnegative decimal numbers. WIDTH is a decimal number (can be negative); using a negative WIDTH implies reversed significance. BIT_LIST is a list of nonnegative bit numbers separated by commas.

The extensions have an implied bit number adjacent to the previous field. If no BIT_FIELD_DESCRIPTOR exists in a field definition, the field is assumed to be one bit wide and adjacent to the previous field. The formats that have implied starting bits are advantageous because it allows you to insert fields into the format specification without having to edit all of the bit numbers that changed.

The following example shows a complete format definition:

```
format ipu_format
  {
        state < 407 .. 404 >:ipu_state;
        red_led: < 104 >:on_off;
        green_led*:on_off*; /* adjacent to red_led */
  }; /* END format ipu_format ... */
```

Since the *scan* utility also determines ring size from the format descriptions, the top bit of the scan ring must be referenced, even if never used.

3.2.4 Ring Specification

You must define each physical scan ring that is to be interrogated or modified during a scan session using the ring specification. Format and ring definitions are separated because there are cases where two boards (rings) have the same format. The ring specification identifies:

- The symbolic name of the ring.
- The bit number (logical address) of the ring in the CMR.

- The symbolic name of the format associated with the ring.

It has the form:

ring *RNGNAME < BIT_NUMBER >:FORNAME;*

As the example illustrates, begin the ring specification with the word *ring* followed by the symbolic name (RNGNAME), followed by the bit number (BIT_NUMBER) contained in angle brackets (< >). The bit number is a decimal number in the range of 0–31. Follow the bit name with a colon and the name of the format (:FORNAME); terminate the specification with a semicolon.

An example of a ring specification follows:

ring ipu < 20 >:ipu_format;

3.2.5 Screen Specification

You can define any number of screens to be edited with the interactive screen editor. Each screen requires a screen specification that identifies:

- The symbolic name of the screen.
- The fields to be displayed on the screen.
- The row and column of the screen where each field is to be displayed (optional).
- The format(s) used to display each field.

Each screen definition begins with the word *screen*, followed by the symbolic name of the screen, and a list of display field descriptions. Any one screen may contain fields from several different rings. Therefore, you may want to define screens with logically associated fields from different boards, so that the interaction among the boards can be monitored on a single screen with respect to a logical function.

The following example illustrates the screen specification format:

screen *SCRNAME*
{
RNGNAME:FIELDNAME [< ROW, [COL] >] :FORMSPEC;
}; /* END screen SCRNAME */

As the example shows, the ring fields are identified by the ringname, followed by a colon, followed by the fieldname (in the associated format specification). You can include an optional row or column specification that specifies where on the screen the field is to be displayed. If you omit this, the compiler automatically selects a convenient place to put it. If present, you must specify the row in the range 1–20. You are allowed to omit the column, which indicates the display can be anywhere on the specified row.

The format specification (FORMSPEC), which specifies the form to display the field value, consists of a string of any combination of *s*, *h*, and *b*. Specifying *s* indicates that the field value is to be displayed in symbolic form (according to its synonym list specification). Specifying *h* indicates hexadecimal and *b* indicates binary. You can specify that a field be displayed in more than one format. The following example shows a screen specification:

```
screen general
  {
        ipu:red_led : sb; /* display anywhere in symbolic and binary */
        ipu:green_led < 5, > : sb; /* display on row 5 */
        ipu:state < 10, 10 > : h; /* display on row 10, column 10 in hexadecimal */
  }; /* END screen general */
```

3.3 Scan Compiler

The scan compiler translates the scan definitions file into a binary file, which is used as input to the interactive scan utility.

> **NOTE**
>
> Because of the differences of the internal representations of data on different machines, it is required that a binary file supplied to the interactive scan utility be the product of the scan compiler on the same machine. THE BINARY FILE DOES NOT TRANSPORT ACROSS MACHINES. THE TEXT FILE DOES.

The scan compiler reads the definitions from its standard input file, and if no errors occur, places the binary into the standard output file. If errors occur, a message is written to the standard error file, and the standard output file contains a copy of the definitions file with error messages placed appropriately.

To invoke the scan compiler, enter:

scanc *definition_filename*

The compiler then reads the definitions file (*definition_filename.scd*), writes to *definition_filename.sco*, and sends errors to *definition_filename.scev*.

3.4 Scan Utility Operation

After you created and compiled the definitions file and all errors have been corrected, you invoke the interactive *scan* utility as follows:

scan *filename*

where *filename* is the binary output from the scan compiler. (This causes the *scan* program to read the definitions file, i.e., *definition_filename.sco*).

After *scan* initializes, you are prompted for input by:

scan:

You are in the line-oriented mode of *scan*. At this point, you can enter the appropriate commands or enter the name of your scan script file. *Scan* will then execute the commands or the script file. (For information on writing a script file, see "Scan Script File Format.")

You can also display the following command summary, by entering **help** or **?** at the *scan* prompt.

Figure 3–1, CONVEX Scan Utility Command Summary

```
name      alias      meaning          name        alias        meaning
                                      |
help        ?     help                |loadram       lr     apply load ram pulse
put         p     put field value     |loadscan      ls     apply load scan pulse
get         g     get field value     |clock         c      apply clocks(s)
read        R     read ring(s)        |run           r      run board(s)
write       W     write ring(s)       |iupdate       iu     set update flag
reset       re    reset subsystem(s)  |verify        v      set verify flag
log         l     create log file     |esr           E      display esr register
logl        ll    create log file     |cgr           C      display cgr register
execute     x     execute file        |sh            !      execute shell
executel    xl    execute file        |allbits       ab     list ring bit fields
edit        e     interactive edit    |bit           sb     list single bit field
editl       el    interactive edit    |all           a      list all ring values
snapshot    sn    log ring values     |screens       sc     list screen names
snapshotl   snl   log ring values     |print         pr     print string
putlog      pl    put string to log   |
putlogl     pll   put string to log   |

for more information type:

       help <command name>
```

3.4.1 Scan Utility Commands

The following paragraphs describe the commands that you can use with the *scan* utility. You can display these commands with the self-help facility of scan by entering **help** at the *scan* prompt.

help (?)

Use the *help* command to display a command summary. If you want information for a specific command, follow *help* with the command name.

FORM: **help** [command name]

EXAMPLES: **help**

?
help snapshot

put (p)

The *put* command places the specified value in the specified field. You can specify a specific format (h, s, b—described in "Screen Specification") by following the fieldname with a slash and the format descriptor. If you do not indicate a format specification, the format defaults to hexadecimal.

If you have set the *iupdate* flag to *on*, using the *put* command causes an immediate write to the hardware ring. However, if the flag is *off*, only the local copy of the scan ring is updated; the hardware write then occurs only when you execute a *write* or *iupdate on* command.

FORM: **put** *fieldname hex value*
put *fieldname* **/b** *binary value*
put *fieldname* **/s** *synonym value*
put *fieldname* **/h** *hex value*

EXAMPLES: **put mcu:red_led /s on**
p mc:red_led /b 2#001
p mc:green_led 16#ffff

get (g)

Use the *get* command to display the specified field value in hexadecimal and as a synonym, if one is defined. This command, which never results in a hardware read, takes the value of the field from the local copy of the ring data.

FORM: **get** *fieldname*

EXAMPLES: **get mcu:red_led**
g mcu:red_led

read (R)

Use this command to read the indicated (or implied) hardware scan rings into *scan's* local scan ring data copies.

Scan maintains a local copy of scan ring data for each defined scan ring. When *scan* starts up via an implied *read* command, these copies are initialized and used throughout the scan session for *get* and *put* commands. An implied read (after each scan ring hardware write for verification) occurs when the *verify* flag is set to *on*. If you do not want an implicit read, set the *verify* flag to *off*.

Using the *read* command without the ringname(s) reads all defined scan rings.

FORM: **read** [*ringname . . .*]

EXAMPLES: **read ipu mcu**
read
R

write (W)

The *write* command causes the indicated (or implied) hardware scan rings to be written from *scan's* local scan ring data copies.

Scan maintains a local copy of scan ring data for each defined scan ring. Using the *put* command with the *iupdate* flag set to *on* causes an immediate write the the hardware scan ring. When the *iupdate* flag is *off*, execution of the *put* command causes the local copy of the scan ring data to be updated; no hardware write takes place. If you set *iupdate* to *off*, hardware writes occur only through the explicit execution of the *write* command.

When the *verify* flag is *on*, after each implicit or explicit hardware write, a read/compare operation is performed. However, whenever the flag is *off*, no implicit reads occur.

Using the *write* command without the ringname(s) writes all defined scan rings.

FORM: **write** [*ringname* . . .]

EXAMPLES:
write ipu mcu
W ipu
write

reset (re)

Use the *reset* command to reset the specified subsystem of the CONVEX computer. The reset is performed by applying 16 clocks to the subsystem boards with the appropriate bit set in the System Reset Register (SRR).

The only time resets occur is upon explicit execution of the *reset* command.

If you do not indicate a subsystem(s), then all subsystems are reset.

FORM: **reset** [**jp** | **hsc** | **mem** | **mult**]

EXAMPLES:
reset jp
reset
re

log (l), logl (ll)

These commands allows you to establish a file in which commands are written as they execute. Then you can execute the log file to recreate the scan session.

The commands *log*, *execute*, *execute*, *edit*, and *snapshot* do not get logged, as their execution at a later time would not reproduce the original effect. However, you can use the alternate forms for these commands, (i.e., *logl*, *executel*, *editl*, and *snapshotl*), which will get logged and executed, if the log file is executed.

Initially no log file is established. If you do not include parameters with the *log* command, the current log file (if any) closes at execution. Thus, establishing no logging of commands. To establish a log file, you must follow the command with the filename (indicating the log file).

FORM: **log** [*filename*]

EXAMPLES:
log setup.log
l setup.log
ll setup.log
l
ll
log

Execute (x), executel (xl), execute e (xe), execute s

These commands cause the command interpreter to begin executing commands from a specified file. The command interpreter continues executing commands from the file until end of file is reached, at which time it returns to its previous source for commands. One command file may have an *execute* command for another command file. The command file may be a previous log file or an edited file. If you use the *execute* (*x*) form of the command, then this command itself is not logged. If you want this command to be included in the log file, use the form *executel* (*xl*). You can also execute a script file by entering only the script filename at the *scan* prompt.

If you want to echo the commands to the terminal as they are executed, use the *e* option. Use the *s* option if you want to print one command and then pause. If you use *x s filename*, *scan* executes the first command, prints the command, and then pauses. To continue, you must strike the RETURN key; then *scan* executes and prints the next command. To cause *scan* to execute a specific number of commands, print them to the screen, and pause, you can type *n* spaces before the return. For example, if you want to execute five commands, enter **x s foofile<CR>**. Then when *scan* pauses, strike the space bar five times followed by a carriage return. *Scan* will execute the next five commands.

To abort this session, use CTRL C.

FORM: **execute** [*e,s*] *filename*

EXAMPLES:
execute setup.log
x setup.log
x e setup.log
x s setup.log
setup.log

edit (e) and edit (el)

You can use either of these commands to invoke the interactive screen editor (*edit* is not logged; *editl* is logged). The screen editor simplifies the process of interrogating and updating scan ring values, by allowing you to peruse and edit fields of a predefined screen. The editor acts as a command generator to the rest of the *scan* utility. For example, when a displayed field is modified, the editor generates a *put* command, displays it at the bottom of the screen, and passes the command to the scan command

line interpreter for execution. Therefore, the functional capabilities of the editor compose a subset of the functional capabilities of the line-oriented command set.

The editor generates the following commands: *put*, *read*, and *write*. If a log file is established, these commands are logged as they are generated. Thus, you can reproduce the effects of an edit session later by executing the log file. Use **^?** for help from the editor.

FORM: **edit** *screen_name*

EXAMPLES: **edit state_screen**
e state_screen

To speed the use of the screen edit mode, you can use a screen subwindow. This feature allows for rapid switching between the edit and line modes. After you have started a screen edit, you can move to the subwindow by entering **CTRL E**. After you enter *CTRL E*, the scan prompt appears at the bottom of the screen. The bottom five lines of the screen are erased and all commands entered will scroll the bottom five lines only.

To exit the subwindow, enter **CTRL E** followed by a carriage return.

Use of the subwindow does have some limitations. The edit screen works fine for the new screen; however, when returning to the original screen the two screens write on top of each other. The *quit* command from the subwindow leaves the terminal in a mode where all scrolling is done on the bottom five lines of the screen. Thus, making it necessary to reset the terminal to restore full screen scrolling. Of course, you can restore the scrolling by reentering the subwindow mode of scan and exiting with the *CTRL E* carriage return sequence.

The data on the top part of the screen is not updated until the subwindow mode of operation has been exited. When the subwindow has been exited, the condition of the scan ring is displayed on the screen.

snapshot (sn), snapshotl (snl)

When you execute either of these commands, they cause *put* commands to be generated and logged in the current log file, making a snapshot of the specified (or implied) ring values. (*Only* snapshotl is logged.) A *put* statement is generated for each field of each specified ring with the current value of that field. When you execute this log file at a later time, the scan rings are restored to their values at the time the snapshot was made.

If you execute *snapshot* without parameters, all defined rings are snapshot.

FORM: **snapshot** [*ringname* . . .]

EXAMPLES: **snapshot mcu ipu**
snl ipu
sn

putlog (pl), putlogl (pll)

Using either of these commands causes all operands of the command to be placed into the current log file; only *putlog* is logged. If you execute the command from the keyboard, no parameter substitution occurs, causing the operands to be placed in the log file as is. However, executing the command from a file using the *execute (x)* command causes parameter and field substitutions to occur before the operands are placed into the log file. Thus, the new parameters and field substitutions are logged.

FORM: **putlog** *parameters*

EXAMPLES:
```
putlog print The value of $1 is ${$1}.
putlog putlog put $1 ${$1} # snapshot field $1
```

loadram (lr)

This command causes a single load RAM pulse to be generated for a specified set of boards. Use the *d* option to indicate the DMODE line. Then DMODE is asserted during the load RAM pulse.

If you do not specify the ring(s), then all defined rings are selected.

FORM: **loadram** [d] [*ringname . . .*]

EXAMPLES:
```
loadram d ipu mcu
lr
loadram ipu
```

loadscan (ls)

This command causes a single load scan pulse to be generated for a specified set of boards. Use the *d* option to indicate the DMODE line. Then DMODE is asserted during the load scan pulse.

If you do not specify the ring(s), then all defined rings are selected.

FORM: **loadscan** [d [*ringname . . .*]

EXAMPLES:
```
loadscan d ipu mcu
ls ipu mcu
loadscan
```

clock (c)

You can use this command to generate a specified number of clocks for a specified set of boards. Use the *d* options to assert DMODE during the clocks. If you do not specify the number of clocks, then a single clock is generated.

FORM: **clock** [d] [*number*] [*ringname . . .*]

EXAMPLES:
```
clock d
c
clock 10000000 ipu
```

clock d 10000000 ipu

To interrupt the execution of a lengthy clock command, use *CTRL C* or *<CR>*.

run (r)

Use the *run* command to place a specified set of boards into a run state. Specifying no rings selects all rings. If you need to interrupt the execution of a *run* command, use *CTRL C* or *<CR>*.

FORM: *run* [*ringname . . .*]

EXAMPLES: **rn ipu mcu**
r
run

iupdate (iu)

The *iupdate* command works in conjunction with the *put* command. You must set the *iupdate* flag to *on* to perform a scan write each time a scan ring field is updated using the *put* command. This causes the scan ring to be written immediately with the new field value. However, if you have the flag set to *off*, executing the *put* command updates only the local copy of the scan ring; the actual hardware is not affected. Executing an *iupdate on* also causes immediate writes of any rings that have been modified since the last read or write.

Using the command without parameters defaults to *on*.

FORM: **iupdate** [**on**|**off**]

EXAMPLES: **iupdate**
iu
iupdate on
iupdate off

verify (v)

The *verify* command allows you to indicate if a read or compare verification is to be done after writes to the hardware scan rings. When you set the flag to *on* or specify no parameters with the command, a read and compare is performed following each scan-ring write. Then, when an error is detected, an error message displays that includes the hexadecimal number, indicating the bits that differed between the written value and the read value.

FORM: **verify** [**on**|**off**]

EXAMPLES: **verify**
v
verify on
verify off

esr (E)

You can use this command to display the source register.

FORM: **esr**

EXAMPLES: **esr**
E

cgr (C)

You can use the command to display or update the Clock Gating Register (CGR). To display the current value, you enter no parameter. Follow the command with the number to set the clock gating register.

FORM: **cgr** [*hex number*]

EXAMPLES: **cgr 0010000**
cgr
C

sh (!)

Use this command to execute a shell. To terminate the shell and return to *scan*, use **CTRL D**.

FORM: **sh**

EXAMPLES: **sh**
!

allbits (ab), bits (b), all (a), screens (sc), print (pr)

All of these commands have display capabilities as described in the following table:

Table 3–1, Display Capabilities of Scan Commands

Command	Description	Form	Usage
allbits (*ab*)	Displays bit field descriptor for all fields of specified ring	**bit** *ringname*	**allbits mcu** **ab mcu**
bits (b)	Displays bit field descriptor for specified field	**bit** *ringname:fieldname*	**bits mcu:red_led** **b mcu:red_led**
all (a)	Displays value of all fields in specified ring	**all** *ringname*	**all mcu** **a mcu**
screens (sc)	Displays names of all defined screens	**screens**	**screens** **sc**
print (pr)	Displays assigned variable value or string	**print** *variable/string*	**print red_led** **pr :01** **pr A string**

3.5 Scan Script File Format

The scan scripts are comprised of *scan* utility commands grouped by control flow statements and various assignment statements added to support internal variables.

The language that you can use in the script file is fashioned after the Cshell scripts. However, the control flow language (see "Scan Control Flow Language Structure" for more information) has a different syntax. You are limited to a maximum of 32-bit fields in variable expressions. However, variables that are used as temporary storage can contain longer strings as long as the variables with long strings are not used in equations. Since the scan script is written using scan utility commands, the only way the hardware interface implementation can be modified is to change the scan utility program.

In your script file, you can use labels and statements that are written one to a line in the file, blank lines, and comment lines, if necessary. No line may contain more than one label or statement. Designate a comment line with a #. You are not allowed to use statement lines of more than 32 fields. Exceeding 32 causes an error.

A script might appear as:

```
#This is a sample
#of the body of a scan script.
.
.
.
statements    #explanation of statement, if needed
```

If you have made format errors in your scan script, an error message prints during execution of the script.

You can add parameters to the command lines that cause execution of the script. The scan script can then access the parameters by using a format similar to the variable format. The parameters can be accessed by a $n where *n* is the parameter number starting from 1.

To help in parameter processing, you can use the *?n* format. The *?n* converts to 1 or 0 depending upon whether a parameter was passed or not passed, respectively. You can use the *?n* format in computational expressions where an internal variable can be used.

3.6 Scan Control Flow Language Structure

The scan control flow language, which is used to write scan scripts, has structures that support the following areas of control:

- Internal variables
- Test conditionals
- Looping and branching structures
- Miscellaneous statements

The control language functions as follows. Each command line read in is first scanned for macro parameters and then for internal variables. All values are substituted prior to the execution of the command line. The control flow lines are turned into comments prior to being passed to the command processor. Consequently, the control flow language statements are only valid when read in from a command file.

For example, in the following scan script, command lines are read for macro instructions, bits are set, 5 clocks are set, and the *psw* is read out of the *asu*. Use of the IF statement allows the value of *psw* to be printed or to be called from a script.

```
p asu:uword 830ca1f70e91df004700
p asu:halt_disab* 0
p asu:frc_ui_lvl* 0
c 5 asu
p asu:frc_ui_lvl* 1
if :98 == 0
  pr PSW - ${asu:bscan}
else
  return ${asu:bscan}
endif
```

3.6.1 Internal Variables

The scan control flow language allows you to use internal variables in conditional statements. These variables are of the form:

```
:xx
```

where *xx* is 00 to 99 inclusively.

You can use these variables in two ways. The first is a general purpose variable; the second, a computational variable. The value of the variable is always a 32-bit hexadecimal integer value.

3.6.1.1 Computational Variables

The ASSIGN statement initializes the variable to the value defined. This value can be a constant, the result of a *get* field operation, or another variable as well as an expression of constants, variables, and fields.

The variables 00 through 97 are general purpose registers and can contain 32-bit hex values stored as ASCII strings. You can also use 99 as a general purpose register; however, it is the register that contains the result of a *return* command in a macro called by the current macro. By using the variable 99 as a returned value register, a macro can evaluate the result of another macro's execution. (See "Special Variables," section 6.1.4 for information on variable 98.)

You can use the ASSIGN statement in two forms. The first:

:variable_number = *character_string*

character_string ::= a string of nonblanks terminated by blank, newline, or tab

The following example illustrates correct formats:

Form	Explanation
:10 = 0	# assigns string 0 to variable
:12 = 012345678	# assigns string 012345678 to variable
:13 = Hello	# assigns string Hello to variable
:14 = Hello there	# assigns only string Hello to variable
	# Only the first string of nonblanks
	# is assigned to a variable; all others are ignored

The following example illustrates invalid formats:

Form	Explanation
:10 0	# no equal sign present
:12 =	# no value to assign
= 10	# no variable to assign to
:102 = 0	# invalid variable (it assigns to variable 10)

In the assign expression, you can use any of the following operators:

Operator	Explanation
+	adds two numbers
-	subtracts two numbers
&	includes (and) next two numbers
|	inclusive OR two numbers
^	exclusive OR two numbers
>>	shift right
<<	shift left
~	one's complement of number

Regardless of the operator, all expressions are evaluated from left to right without regard to the type of operator. For example:

```
:10 = 1234 + abcd & 1 +1                    # equals 2
```

Whenever you use an operator in an expression, you must precede and follow the operator with a **blank**, except the operator ~. The operator ~ must be placed against the value to be complemented. (See the examples illustrating valid and invalid assignment expressions.)

The second form of the assign statement:

: *variable_number* = *assign_expression*

assign_expression ::= *value* [+,-,&,|,^,<<,>> value]

value ::= [~]: *variable_number* | [~] *constant* | [~]${*field*}

constant ::= a number in HEXADECIMAL

The following examples illustrate valid formats of the assign expression:

Table 3–2, Valid Formats of the Assign Expression

Form	Explanation
:22 = abcdef	# hex value assignment
:23 = aff - a - a - 3	# negative value assignment
:24 = ${ipu:grn_led}	# field value assignment
:25 = ${asu:status} 7 ~1c	# variation of assignment

3.6.1.2 General Purpose Usages

You can use the internal variables as storage for character strings of various lengths and values. You can store the values of fields longer than 32 bits in a variable without error as long as the variable is not used in computations. Thus, the variables can be used to save the values of a field for later reassignment.

3.6.1.3 Indirect Variable Assignments

The indirect ASSIGNMENT statement allows you to make ASSIGNMENT statements dynamic in a manner similar to the GOTO statement. The indirect ASSIGNMENT statement has the form:

::*xx* = *assign_expression*

The following illustrates a valid usage of the indirect ASSIGNMENT statement:

```
:10 = 03
::10 = :20 + 60
```

These statements assign the result of :20 + 60 to :03.

The contents of the indirect variable are assumed to be decimal and at least two characters long. For example, a hexadecimal 10 assigns the value to the variable decimal 10. You may want to use the *dec* command, (i.e., **dec** :*xx*), to convert the contents of a variable from hex to decimal in GOTO statements or indirect variable assignments.

3.6.1.4 Special Variables

The scan control language provides two special variables.

The first, variable 99, contains the string returned by a called macro. This variable can be used again in any fashion consistent with the value returned.

The second, variable 98, contains the calling level of the macro. The calling level from the *scan* utility is 0, and as a macro calls another macro, the level increases by 1, up to the maximum of 7. You can use variable 98 in any expression as long as the variable contains a valid number.

3.6.2 Conditional Expressions

The scan control language provides the ability to evaluate conditional expressions. You can use any of the following conditional expressions in IF statements only.

Table 3–3, Conditional Expressions in IF Statements

Modifier	Meaning
==	is equal to
!=	is not equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to

If the condition is true, the result equals 1; otherwise, results equals 0. You can use variables, constants, and field values in all conditional expressions.

3.6.3 Looping and Branching Structures

The use of IF and GOTO statements or loops allow you to redirect the flow of a program, as they provide a medium for altering the normal program execution sequence.

3.6.3.1 IF statements

The IF statement has the form:

```
if conditional statement
        .
        . (statement(s))
        .
else
        .
        . (statement(s))
        .
endif
```

where *conditional statement* represents the assign expression and conditional modifier preceding an optional assign expression; that is:

conditional statement::=assign_expression [condmod assign_expression]

The IF statement redirects control depending upon the conditional value. If the conditional value does not equal 0, then the first (IF) statements execute. However, if the conditional value equals 0, then the ELSE statements execute.

The following examples illustrate valid formats of the IF statement:

```
if:10                                   # value compared to zero
  .
  . (statements)                        # executed if not zero
  .
endif                                   # must be present


if:10 + 1 === :20 & 1e                  # value
  .
  .(statements)                         # executed if equal
  .
else
  .
  .(statements)                         # executed if not equal
  .
endif                                   # must be present
```

You must always include an ENDIF; otherwise, the macro either finishes too early or skips unexpected blocks of the macro file. In addition, you must have included a computational expression as part of the IF statement. The IF statement cannot be used to compare two strings that are not 32-bit hexadecimal integer expressions. You can, as the example shows,

use a complex expression for the conditional expressions.

3.6.3.2 GOTO Statements

The GOTO statement allows you to transfer control to a specified statement. You can use a GOTO statement with the IF statement for simple looping. During statement execution, control transfers to the statement identified by the statement label. The object of the GOTO must be a valid label in the form:

L*xx* # comment

where *xx* is 00 to 99. The comment line is optional. You can use label lines in any position in the file. It is possible to jump to a label inside the same FOREACH loop. However, using GOTO's out of a loop is not a good practice as this leaves internal structures in a manner that may inhibit future FOREACH loops.

You cannot follow the label with a *scan* statement. If you do, the *scan* statement does not execute. Thus, a statement as follows:

L10 p ipu:red_led 1

would not execute as the label is followed by a *scan* statement. A comment is optional and does not effect execution.

The defined label can be reached by a GOTO statement:

goto L*xx* # comment

3.6.3.3 FOREACH Statements

Using a FOREACH loop allows you to place a set of values in a variable one at a time; then execute a set of statements. When a FOREACH statement is reached, the values assigned are resolved for their values at that time. Each time a pass occurs, a value from the list of resolved terms is assigned. The general format for the FOREACH statement is:

foreach *:variable_number set_of_values [comment]*

set_of_values ::= value [value ... value]

value ::= :variable_number | *constant* | ${*field*}

The following example illustrates a valid format of the FOREACH statement:

```
foreach :10 1234 :23 abcd ${ipu:red_led}
.
. (statements)
.
foreach :44 s0 s1 s2 s3 s4 :23 ${asu:status}
   .
   .(statements)
   .
```

```
    end
end
```

You must always supply the END statement as the loop cannot loop without it.

As the example shows, you are allowed to nest loops. When you use nesting, be sure the loops do not overlap ranges; each loop must be contained in the previous FOREACH loops. You can use a maximum of eight levels of nesting.

The control language provides the BREAK statement to allow a FOREACH loop to be escaped early. When a line is read in and the first string on the line is the word *break*, the current loop is exited. The format of the BREAK statement is:

```
break    # comment
```

3.6.4 Miscellaneous Statements

The following are miscellaneous statements:

3.6.4.1 COMPARE Statement

The scan control language provides the COMPARE (cmp) statement an as aid in evaluating parameters and synonyms. The results of a COMPARE statement can be tested by the IF statement. It has the form:

cmp *:xx string_constant string_constant [comments]*

where *xx* is the variable where the results are stored and *string constant* can be a literal constant, an internal variable, a parameter, or a field value.

The following examples illustrate the formats of the COMPARE statement:

```
cmp :10 hello Hello    # comment
```

In this example, :10 would contain 0, indicating not equal.

```
cmp :10 test test      # comment
```

In this case, :10 would contain 1, indicating equal values.

3.6.4.2 RETURN Statement

You can use the RETURN statement to return a character string to the calling macro. The calling routine can access the value returned through the special variable 99. This special variable (99) can be used in any format that its value allows. If a character string is returned that is not a hex number, using variable 99 in computational expressions then causes an error.

The RETURN statement has the form:

return value

Chapter 4
Diagnostic Utilities

4.1 Overview

This chapter describes publicly accessible diagnostic commands in alphabetical order.

THIS PAGE INTENTIONALLY LEFT BLANK

NAME

intro – introduction to commands

DESCRIPTION

This section describes publicly accessible diagnostic commands in alphabetic order. The word 'local' at the foot of a page means that the command is not intended for general distribution.

NOTE

All diagnostic tests within this chapter are off-line in nature and SHOULD NOT be executed while CONVEX UNIX is running.

CROSS TOOL REFERENCES

References are made to cross tools and libraries that were used in the development of the service unit processor SPU software. The cross tools are available only on the diagnostic development system and are not available under SPU UNIX. The cross tool information is included for those doing diagnostic software development. Cross tool references are indicated by lists of files preceded by the word **HOST:** under the **HOST LIBRARIES,** or **SEE ALSO** sections, or by the section **HOST LIBRARIES.** File lists not preceded by the word **HOST:** specify files that are available under SPU UNIX.

DIAGNOSTICS

Upon termination each command returns two bytes of status, one supplied by the system giving the cause for termination, and (in the case of 'normal' termination) one supplied by the program, see **wait(2)** and **exit(2)**. The former byte is 0 for normal termination, the latter is customarily 0 for successful execution, 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.

NAME

boot_iop – program to coldstart boot an IOP or VIOP and download an object file to it

SYNOPSIS

boot_iop -Xrtq object_file

DESCRIPTION

Boot_iop will initiate a coldstart boot of the selected IOP or VIOP.

The options to the program are:

X is the CCU number to load. The only valid CCU numbers are 3 through 7.

r will signal that the object is to be loaded at the entry point in the object file, but the CCU will simulate a reset interrupt to enter the code.

t will indicate that the IOP will be commanded to execute the EPROM selftest code before any load is started. This option is not available for VIOP's.

q will suppress any "normal" information printouts, but will not stop the error message printouts.

NAME

config_chk – verify CPU configuration

SYNOPSIS

config_chk

DESCRIPTION

Config_chk is used on power up to verify that there is a valid CONVEX-1 CPU configuration present. **Config_chk** performs its validation process using the data base file: */mnt/usr/lib/DB_config*. This data base file contains the board revisions and the micro-code revisions which comprise all the supported CPU configurations. The format of the file is as follows:

```
config: XXX  id: YYY
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  BOARD_ID  RING_REV  PN.ASM_REV(S)
  UCODE_ID  MAJOR.MINOR
  UCODE_ID  MAJOR.MINOR
  UCODE_ID  MAJOR.MINOR
  UCODE_ID  MAJOR.MINOR
config: XXX  id: YYY
    :        :         :
    :        :         :
```

The valid BOARD_ID's are: **asu**, **atu**, **ipu**, **mcu**, **pcu**, **vcu**, **vpu0**, and **vpu1**. The valid UCODE_ID's are: **base**, **actl**, **lctl**, and **mctl**. Both the PN.ASM_REV(S) and the number of configurations can vary in number from one to ten.

Config_chk checks all the configurations of the data base file, accumulating match/mismatch information for each entry. It then checks if a valid configuration was found. If so, it outputs the configuration name and ID to both *stdout* and to the file */mnt/usr/lib/config* for future reference by other programs. If a valid configuration is not found, **config_chk** determines the best fitting configuration and outputs that configuration to *stdout* noting miscompares with asterisks. **Config_chk** also leaves the file */mnt/usr/lib/config* as a null file to indicate to future programs that no valid configuration exists.

FILES

/mnt/usr/lib/DB_config
/mnt/usr/lib/config

DIAGNOSTICS

Config_chk will report the following errors:

```
Valid system configuration not found
Unable to open data base file
Board or micro-code entry format error
Micro-code file doesn't exist or has wrong magic number
```

NAME

cop – display board identification information

SYNOPSIS

cop [-vem] slot ...

DESCRIPTION

With no options, **cop** displays the slot name, board type, part number, serial number, and board revision of the boards in the **slots** specified. **Cop** also verifies proper board/slot matching. If a mismatch is detected, an error message is printed indicating the mismatch, and **cop** exits with a return status of -1. Otherwise, **cop** returns a status of 0.

One or more of the following slot mnemonics may be used with **cop:**

all	cpu, mau, and io slots
cpu	All CPU slots plus MCU and SPU
mau	All MAU slots
io	All I/O slots
asu	ASU slot
atu	ATU slot
ipu	IPU slot
mcu	MCU slot
pcu	PCU slot
spu	SPU slot
vcu	VCU slot
vpu0	VPU0 slot
vpu1	VPU1 slot
ccu1	CCU1 slot
ccu2	CCU2 slot
ccu3	CCU3 slot
ccu4	CCU4 slot
ccu5	CCU5 slot
ccu6	CCU6 slot
ccu7	CCU7 slot
mau0	MAU0 slot
mau1	MAU1 slot
mau2	MAU2 slot
mau3	MAU3 slot
mau4	MAU4 slot
mau5	MAU5 slot
mau6	MAU6 slot
mau7	MAU7 slot
hiaX_1	HIA board 1 attached to HSP in ccu slot X [3-7]
hiaX_2	HIA board 2 attached to HSP in ccu slot X [3-7]
hiaX_3	HIA board 3 attached to HSP in ccu slot X [3-7]

If the *-v* option is specified, board/slot matching is verified, but board ID information is not displayed.

The *-e* option causes information about the manufacturing/test level and missing assembly revisions to be added to the standard board ID information.

The *-m* option displays only the manufacturing/test level code for the named slots. No other information is displyed. This option is intended for use in scripts.

FILES

/mnt/usr/lib/DB_cop

NAME

cpureg – display C1 central processor nonvector register state

SYNOPSIS

cpureg

DESCRIPTION

Routine **cpureg** calls **reg_dump(3D)** to scan and display the contents of **C1** central processor non-vector register state to *stdout*. The following registers are displayed:

pc	program counter register
ca	current address register
psw	program status word register
jpcr	job processor control register
upc	micro program counter register
a0-a7	address registers
s0-s7	scalar registers
t0-t7	temporary registers

NAME

diaginit – diagnostics initialization script

SYNOPSIS

.diaginit [-f]

DESCRIPTION

Diaginit is a shell script invoked at boot time by SPU UNIX to initialize the scan ring description files. If the *-f* switch is used, initialization is forced. Otherwise, initialization only occurs if both the power has been cycled off, and the configuration of boards within the system has changed.

The power-up bit is examined in the soft front panel, and the **cop(1D)** utility is used to determine the set of boards currently installed in the system. If the power-up bit is set and the board set has changed, initialization proceeds. First, **scnlink(1D)** is invoked once to create the diagnostic scan ring description file, */mnt/usr/scn/scn_rings*, and again to create the description file for non-scan-ring boards, */mnt/usr/scn/scn_rings2*. Then, a new */mnt/jptest/jp.sco* file is generated to support the interactive scan utility, **scan(1D)**.

To insure that the CPU control stores will be loaded by **initall(1D)**, the file */mnt/usr/lib/Initall_cpu* is removed if the power-up bit is set or if the *-f* switch is used. Finally, if **diaginit** completes successfully, the power-up bit is cleared.

SEE ALSO

cop(1D)
initall(1D)
scan(1D)
scn(3D)
scnlink(1D)

FILES

/mnt/usr/scn/scn_rings
/mnt/usr/scn/scn_rings2
/mnt/usr/lib/Initall_cpu

NAME

dshell – C1 test executive (diagnostic shell)

SYNOPSIS

dshell

DESCRIPTION

The C1 diagnostic shell, or **dshell**, runs on SPU version 7, UNIX when C1 central processor UNIX is not running. All C1 diagnostics, except standalone and on-line diagnostics, run under this shell. This allows the user to become familiar with one command interface and set of command options. The **dshell** supports two modes of testing: manual mode and automatic mode. Manual mode provides low-level pass/fail information, while automatic mode provides isolation of detected faults to a field-replacable unit (FRU).

COMMAND SYNTAX (Manual Mode)

test	display test menu
test testname	execute test 'testname'
test testname -s	display subtest menu, prompt user for subtests
test testname -s n(i,j-k)	execute subtests i,j-k n times
test testname -c	display class menu, prompt user for classes
test testname -c n(i),j-k	execute classes i n times & j-k
pause off (-f) (-b) (-e)	disables all or specific pauses
pause -f	pause on all failures
pause -f nnn	pause on failures in subtest nnn
pause -b	pause at beginning of all subtests
pause -b nnn	pause at beginning of subtest nnn
pause -e	pause at end of all subtests
pause -e nnn	pause at end of subtest nnn
continue (or carriage return)	continue test from pause
msgs off (-f) (-s) (-t)	disables all or specific messages
msgs -f (long/short)	enable long/short fail messages
msgs -s	enable subtest result messages
msgs -t	enable test result messages
log off (-s) (-t)	disables multiple fail modes
log -s nn	allow nn failures per subtest
log -t nn	allow nn subtest failures per test
loop off (-s) (-t)	disables loop modes
loop -s nnn	loop on subtest nnn
loop -t	loop on test
status	print test flag status
exit	exit from dshell
quit	exit from dshell after all executing and/or queued tests have finished execution
^C	command level is restored to the user if no tests are executing. If any tests are running, a menu of abort procedures will be displayed.

^B	aborts dshell and all executing or enqueued tests
any command -h	print help information on command
help	print help information for all dshell commands
!	executes any UNIX command that immediately follows

COMMAND SYNTAX (Automatic Mode)

fits	enter FITS, the Fault Isolation Test Suite.
msgs -ls n	set screen messages to level *n*.
msgs -lf n [filename]	set file messages to level n, using file *filename*
test testname -a	execute test *testname* in automatic mode.

COMMAND EXPLANATIONS (Manual Mode)

The **test** command causes a test to be executed. If no testname is given, the user is shown a menu of available tests and is 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. The subtest option **-s** and class option **-c** allow for selective, multiple and/or sequenced execution of available subtests and classes. If alone in the command line, they cause a menu of all subtests or classes in the chosen test to be displayed. The user is then prompted for his choices. Correct 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. Examples follow:

-s	display subtest menu
-c	display class menu
-s 10,15,5(20)	executes subtests 10,15,20,20,20,20,20
-s 3(10,15)	executes subtests 10,15,10,15,10,15
-c 2(13-10)	executes classes 13,12,11,10,13,12,11,10

The **-c** and **-s** options may not be mixed. Subtests or classes which are listed but do not exist will be ignored. When responding to a menu, the appropriate option (**-s** or **-c**) is assumed, and should not be included in the input stream. Test input and output may be redirected via the command line as follows:

<filename	test input from file "filename"
>filename	test output to file "filename"
+>filename	test output to both file "filename" and terminal

The **pause** command enables pauses at specific points in a test. The **-f** option causes a pause on every fail if alone, or at every fail in a specified subtest. The **-b** option causes a pause at the beginning of every subtest if alone, or at the beginning of a specified subtest. The **-e** option causes a pause at the end of every subtest if alone, or at the end of a specified subtest. When an executing test reaches a pause condition, it halts execution and prompts the user with the **dshell** prompt ": ". The user may then enter any SPU UNIX command via the "!" directive or any **dshell** command. To continue test execution, the continue command (carriage return) should be entered in response to a **dshell** prompt. The default pause status is all pauses off.

The **continue** command is a *nop* if not executed at a test pause.

The **msgs** command allows the user to adjust the pass/fail messages output by test programs. The **-f** (long/short) option enables either long or short error messages. The **-s** option enables subtest/class result messages. The **-t** option enables test result messages. The default for these flags is long fail, subtest, class, and test messages turned on.

The **log** command allows the user to enable/disable multiple failures. The **-s** option enables the specified number of multiple failures per subtest. The **-t** option enables the specified number of multiple failures per test. The default for these flags is all multiple failures turned off.

The **loop** command allows consecutive execution of a test or subtest. The **-s** option enables consecutive looping on the specified subtest. The **-t** option enables consecutive looping on a test. **^C** functions the same as in *exit*, but does not reset either of these flags. The default for these flags is all looping turned off.

The **status** command displays the current condition of all **dshell** test control flags on the screen, and describes the effect each has in its current configuration.

The **exit** command causes termination of all paused tests and an exit from the **dshell**. The **quit** command waits for any currently executing or queued tests to complete execution, and then exits the **dshell**. **^C** will restore command level to the user if no subtests are being executed. If any subtests are running, a menu of abort procedures is displayed. **^B** aborts the currently executing test and quits the **dshell**. There is a another important difference between **exit**, **quit**, **^C** and **^B**. Commands **exit**, **quit** and **^C** all allow an executing test to finish executing *protected code*. **^B** kills the test regardless of the type of code which is currently executing. Thus, a **^B** can leave the machine in an undefined state, necessitating a reboot.

The **help** command gives a brief syntax and command explanation of all **dshell** commands. The **!** directive is an escape to SPU UNIX, and allows for execution of one command which must immediately follow the **!**. An example would be **!mm**.

COMMAND EXPLANATIONS (Automatic Mode)

The *fits* command invokes the Fault Isolation Test Suite (FITS). This uses a combination of functional level and scan/compare level tests to quickly and accurately isolate system faults to a FRU (field replacable unit).

With the single exception of the RAM's on the memory array units (MAU's), the FRU for the C-1 is one of the system boards.

FITS gives you the option of testing either the entire C-1 system, or various subsystems such as: diagnostic subsystem, memory subsystem, I/O subsystem, or the CPU subsystem. (The dshell prompts for the desired subsystem(s).)

Some test programs that run in manual mode can also be executed in automatic mode. The *-a* option of the test command enables automatic mode for tests that support this mode. Consult the documentation for the desired test to determine if automatic mode is available.

There are only two dshell commands used by FITS or by programs executing in automatic mode. These commands allow the user to specify the amount and type of error and status messages, both on the system console and in the FITS log file. The syntax is:

msgs -ls *n*

where *n* is the maximum "level" of message to be displayed on the console.

msgs -lf *n* [*filename*]

where *n* is the maximum "level" of message to be written to the *fits* log file (default is **/mnt/test/FITS_log**).

The possible levels of messages are:

0: no messages
1: dshell-generated messages only
2: level 1 + fault analyzer messages.
3: level 2 + subtest error messages.
4: level 3 + detailed subtest error messages.
5: level 4 + debug messages.

SEE ALSO

libtest(3D)

DIAGNOSTICS

Error messages should be self explanatory. Most deal with errors in user commands. Some deal with real system problems.

BUGS

^C after a test command has been entered, but before the SPU has successfully forked the test process may not kill the test process.

NAME

epcs – load the C1 entry point control store

SYNOPSIS

epcs [-p] [-l] [-v] file

DESCRIPTION

Epcs loads the C1 entry point control store (EPCS) from a writable control store (WCS) image file. *File* specifies the name of the WCS image file to be loaded. The file name extension is determined by the ASU ring revision. Scan ring revisions of less than 100 are designated for the old ASU and have a file extension of *.wcs*, but only the root name of the file should be specified on the **epcs** command line. Ring revisions of greater than 100 are for the new ASU (FASU) and have a file extension of *.fasu*. The WCS image file must be in *b.out* format with the WCS image located in the text segment and a 32 bit checksum located in the data segment. The file may be generated directly from the HOST *microasm .h* output file via the HOST utility **wcsgen(1)**.

When executed, **epcs** halts the central processor, loads the specified file into service processor unit (SPU) memory, performs a checksum verification on the file contents, and then loads the file image from SPU memory to the WCS. After loading is complete, **epcs** reads the content of the EPCS and compares it with the file image in SPU memory. If the EPCS contents fail to compare, a maximum of five errors are logged before **epcs** is aborted. If an error is detected, **epcs** returns an error code. Otherwise, **epcs** returns a 0.

The following options are recognized. If no options are specified, the default is **-lv.** In all cases the checksum verification is performed prior to any other action.

-p Load EPCS image into SPU memory and prompt user with menu for subsequent action. Possible responses are *l* for load, *v* for verify, or *q* for quit.

-l Load only. Do not verify the contents of the EPCS after loading.

-v Verify only. Compare the current contents of the EPCS to the file specified.

FILES

/mnt/bin/initall

SEE ALSO

initall(1D)
HOST: **wcsgen(1)**, **b.out(5)**

NAME

errintd – error interrupt daemon and logger

SYNOPSIS

errintd [–ehs] [–c nn] [–r nn] [–f FILE]

DESCRIPTION

The **errintd** utility monitors a CONVEX C100 Series computer system for hardware error conditions. Three classes of errors are detected by **errintd**: environmental, soft, and hard errors.

Environmental errors are related to the system operating environment. Environmental errors include temperature out of range or loss of airflow.

Soft errors are usually correctable errors (i.e., the error is transparent to the system user). Soft errors include single-bit memory system errors and a variety of MCU detected BUS errors.

Hard errors include parity errors, internal references to non-existent memory, etc. Hard errors always result in the immediate halt of the Central Processing Unit (CPU) and are, therefore, 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 (**mm_sniff(1d)**).

In the normal operating environment, **errintd** is started by the Operating System (OS) SPU utility */mnt/os/prtlog*. All output from **errintd** is time-stamped by */mnt/os/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** will store a maximum of 60 memory chip entries in the softlog file.

Once the softlog file has reached 75% capacity and a burst of errors occur (e.g., 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, **errintd** will continue 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** will log the error and exit. A copy of the last hard error is kept in the file */mnt/hardlog*.

The following options are interpreted by **errintd**:

–e Log environmental errors.

–h Log hard errors.

–s Log soft errors.

Note:

If one of the previous three options is not specified (i.e., **–e**, **–h**, or –s), **errintd** will default to all three types being logged.

–r nn This option specifies the memory sniff rate in Mbytes/day. This option is passed to **mm_sniff(1d)**. The default is 32 Mbytes/day. If a rate of zero is specified or soft errors are not being logged, the sniffer will not be started.

–c nn The **–c** option specifies the maximum softlog size. The value **nn** is the maximum number of failed memory chips on which the softlog will retain information. The default is 60 entries.

-f FILE Sends **errintd** output to *FILE*. By default, output goes to *stdout*.

SEE ALSO

hard_logger(1D)
mm_sniff(1D)
softlog(5D)

NAME

get_defects – read manufacturer's defect map from an SMD drive

SYNOPSIS

get_defects [-v] [-s serial_number] [-d ioconfig_number]
[filename]

DESCRIPTION

The **get_defects** utility reads the defect data recorded by the drive manufacturer on SMD drives and stores the data in an ASCII file that is readable by the disk formatter, *dev4110*.

The following options are supported:

-v If this option is specified, then all writes to the ASCII file are also output to *stdout*.
Default: No output of defects to the display.

-s serial_number
Use this option to enter the drive's serial number. The number can consist of numbers and dashes and must be from 5 to 31 characters inclusive.
Default: Prompted for the serial number by **get_defects**.

-d ioconfig_number
This option selects the entry number for the drive in the */ioconfig* file. Do not specify this option if the entry number is not known for certain.
Default: The */ioconfig* entries are displayed, prompting for a selection.

filename
This option specifies the ASCII file to which the defect data will be written.
Default: Prompted for the filename by **get_defects**.

FILES

/mnt/bin/get_defects
/mnt/bin/lib/get_defects.x00

The format of the ASCII file created by **get_defects** is:

```
# Serial Number: nnnnnn
n DKD-xxx
d cyl hd bcai len
d cyl hd bcai len
```

where **nnnnnn** is the serial number entered. The **#** means the line is a comment when read by *dev4110*. The second line is the name of the drive. For example, DKD-005 is the name of the NEC 2352 SMD drive. For valid drive names, read the man page on **DB_diskfmt(5D)**. Each subsequent line is one defect from the manufacturer's defect data.

SEE ALSO

DB_diskfmt(5D)

NAME

hard_logger – hard error logger

SYNOPSIS

hard_logger

DESCRIPTION

The **hard_logger** utility isolates the source of a hard error. Once a hard error is detected, **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 Central Processing Unit (CPU). Hard errors include parity errors, internal references to non-existent memory, Channel Control Unit (CCU) bus errors, etc.

Upon invocation of **hard_logger**, all system clocks are halted. This is necessary to scan the various scan rings throughout the system.

SEE ALSO

errintd(1D)

NAME

hsputil – hsp register/memory utility

SYNOPSIS

hsputil [HSP number 3..7] [ECHO64 number 3..7]

DESCRIPTION

Hsputil is a utility to display and modify HSP memory locations. The first prameter on the command line is the CCU slot number where the HSP is installed. If the second parameter is specified, the clocks to that CCU slot will also be enabled. When the modify mode is entered the displayed value can be changed by entering the new hex value and hitting RETURN. If no value is entered the next memory location will be displayed. A **q** will terminate the modify and will return to the prompt

COMMANDS

Commands are of the general form:

command parameters

All address values are in hex. A **q** can be used to terminate any command.

m a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents.The locations are displayed/modified one byte at a time.

mb a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one byte at a time.

mw a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one word at a time.

ml a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one longword at a time.

f a1 [a2] value The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fb a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fw a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this occurs on a word basis. If a2 is not specified then only location a1 is changed.

fl a1 [a2] value The contents of memory locations a1 through a2 are filled with value. This occurs on a longword basis. If a2 is not specified then only location a1 is changed.

M a1 disp This command will cause the memory modify mode to be entered. Memory will be 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 This command will copy a block of memory starting at a1 through a2 and place at a3.

rm regname This command will allow the contents of memory mapped HSP based registers to be examined or modified. The contents of the register will be displayed. Entering a [return] will cause the next register to be displayed. Entering a hex value followed by a return
will change the contents of the register if it is a writeable register. Entering a ^ will cause the previous register to be displayed . Entering a **q** will exit this mode. The valid register names are:

```
crl      clock control register
ctra     counter a
ctrb     counter b
fbpr     force byte parity error
chan_reg channel register
fdr      fake diagnostic register
auxtrr   auxiliary trr
pbhdrm   pbus header (ms)
pbhdrl   pbus header (ls)
pbhdr0   pbus header bytes 0 and 1
pbhdr2   pbus header bytes 2 and 3
pbhdr4   pbus header bytes 4 and 5
pbhdr6   pbus header bytes 6 and 7
atuhdrm  atu output reg (ms)
atuhdrl  atu output reg (ls)
atuhdr0  atu output reg bytes 0 and 1
atuhdr2  atu output reg bytes 2 and 3
atuhdr4  atu output reg bytes 4 and 5
atuhdr6  atu output reg bytes 6 and 7
ier      interrupt enable register
ierm     interrupt enable reg (msw)
ierl     interrupt enable reg (lsw)
isr      interupt status register
isrm     interupt status register (msw)
isrl     interupt status register
```

imapcr map control register
pis pbus interrupt status
pic pbus interrupt channel
berrlog bus error log
pberrlog pbus error log
dregberr buffer_in_bus<55..48>
ibpar buffer parity error
addrhi cpuaddr<23..16>
addrlo cpuaddr<15..00>
trr test result register
cdr control diagnostic register
slotid slot id

rd regname This command will display the bits in the named register. Each bit position will be labeled with the function of the bit. Each bit can be individualy modified by moving the cursor over the bit and entering a 0 or a 1. The cursor will move to the right (towards the lsb) when a new bit value is entered or when the space bar is depressed. The cursor will move to the left when a DEL or BS key is pressed. The input is terminated when a RETURN , **q** , ^ or = is entered. A RETURN will update the register and then display the next register. A **q** will exit this mode. Entering a ^ will update the register with the modified value then display the previous register. Entering a = will update the register with the modified value then redisplay the same register.

q[uit] This command will exit **hsputil.**

rhiav This command will dispaly the voltages in an HIA.

? This command will print out a list of valid commands.

NAME

icache – load the C1 instruction cache

SYNOPSIS

icache [-p] [-l] [-v] **ifile**

icache -d ofile [**n1** [**n2**]]

DESCRIPTION

Icache loads and verifies the C1 instruction cache (icache) on the instruction processing unit (IPU). **Icache** also permits the cracked contents of the instruction cache to be dumped to a specified file.

If the first form of the command is used, **icache** loads the object file **ifile** into the service processing unit (SPU) memory, halts the central processor, purges the instruction cache, and then loads the file from SPU memory into the instruction cache beginning at block 0. After loading is complete, **icache** reads the contents of the cache and compares them against the results obtained by cracking the instructions contained in the object file. If the instruction cache contents fail to compare, a maximum of five errors are logged before **icache** is aborted. If an error is detected, **icache** returns a -1. Otherwise, **icache** returns a 0.

In order to crack the instructions contained in the object file, **icache** utilizes information contained in the file */mnt/usr/opcode.hex*. This file must be present before the icache verify operation can be performed.

Ifile specifies the name of the object file to be loaded and to be used for verifying the contents of the instruction cache. **Ifile** must be a CONVEX **a.out** file with an extension of **.o** (magic number of 507). However, only the root name of the file should be specified on the command line.

The following options are recognized by the first form of the **icache** command. If no options are specified, the default is **-lv.**

-p Load object file into SPU memory and prompt user with menu for subsequent action. Possible responses are **l** for load, **v** for verify, or **q** for quit.

-l Load only. Do not verify the contents of the instruction cache after loading.

-v Verify only. Compare the current contents of the instruction cache to the results obtained by cracking the object file.

If the second form of the command is used, the central processor is halted, and the contents of the instruction cache are dumped in cracked format to the output file *ofile*. Blocks **n1** through **n2** of the icache are dumped. If no block numbers are specified, then the entire icache is dumped. If an error is detected, **icache** returns a -1. Otherwise, **icache** returns a 0.

FILES

/mnt/usr/ucode/opcode.hex
/mnt/bin/initall

SEE ALSO

initall(1D)
HOST Unix: */usr/include/a.out.h*

NAME

initall – initialize the C1 control stores and main memory

SYNOPSIS

initall [-cfn] [-p path] [-command arguments] ...

DESCRIPTION

Initall initializes the C1 control stores, the instruction cache (icache) RAMs (optionally), and all of main memory. The control stores are loaded from various microcode files in the directory */mnt/usr/ucode*.

However, the **-p** switch can be used to specify an alternate directory. If an error is detected during initialization, **initall** will abort the initialization sequence. **Initall** returns a status of 0 for successful initialization and a status of 1 if the initialization fails.

Initall with no arguments performs initialization on both the CPU and memory.

The **-c** argument implies conditional initialization of the CPU. In this case, if the file */mnt/usr/lib/Initall_cpu* exists, the CPU is assumed to be initialized, and only memory initialization is performed. If */mnt/usr/lib/Initall_cpu* does not exist, both the CPU and memory are initialized.

The **-f** argument forces the icache to be initialized (if the other control stores are initialized) regardless of the assembly revision of the ATU. Without this argument, the icache will only be initialized if the ATU in the system is older than revision *n*.

Finally, the **-n** switch causes **initall** to print (trace) the initialization sequence without actually initializing any hardware. Note that, in combination with output redirection, this option can be used to generate an initialization script file if necessary for troubleshooting purposes.

The **-command** switch may be any of the following, and can be used to override the default arguments for the command: **-epcs**, **-icache**, **-interleave**, **-margin**, **-mminit**, **-sysreset**, **-vcs**, or **-wcs**. The arguments that follow are arguments to the specified command. For example, to initialize the system with an interleave of 4 instead of maximum interleave (the default), use:

initall -interleave 4

Note that if multiple arguments are to be specified for the command, the arguments must be quoted. For example, to cause initall to initialize the system with non-default margin conditions, use:

initall -margin "-u clk -l ps"

FILES

/mnt/usr/ucode:
actl.acs, base.wcs, base.fasu, init.o, lsctl.lcs, mctl.mcs, opcode.hex, opcode.1135

/mnt/usr/lib:
Initall_cpu

SEE ALSO

epcs(1D)
icache(1D)
interleave(1D)
margin(1D)

mminit(1D)
sysreset(1D)
vcs(1D)
wcs(1D)

NAME

interleave – set/display main memory interleaving factor

SYNOPSIS

interleave [**-h/l**] [**-n**]

DESCRIPTION

The utility **interleave** provides a means for setting and displaying the interleaving factor of main memory in the C1 supercomputer (**NOTE** that interleaves above **4** require MCU 410-1136-200 and PCU 410-1137-200). The **-h** option sets the factor to the highest factor supported by the present hardware configuration, and the **-l** option sets the factor the lowest possible number. The factor may be set to a specific number by using the *-n* option, where *n* is a valid decimal number selected from the list below. Invoking **interleave** with no arguments or invalid arguments displays the interleave that is currently in effect along with a list of the available interleaving factors.

NO. OF MAUs	INTERLEAVE MAX	MIN
1	4	4
2	8	4
3	4	4
4	16	4
5	4	4
6	4	4
7	4	4
8	32	4

DIAGNOSTICS

The current interleave value is returned or -1 if the value cannot be determined. The interleave value cannot be changed while the CPU is running. Error messages are self explanatory.

FILES

/mnt/usr/lib/interlv
/mnt/bin/initall

SEE ALSO

initall(1D)

NAME

ioputil – iop register/memory utility

SYNOPSIS

ioputil [IOP number 3..7]

DESCRIPTION

Ioputil is a utility to display and modify IOP memory locations. When the modify mode is entered the displayed value can be changed by entering the new hex value and hitting RETURN. If no value is entered the next memory location will be displayed. A **q** will terminate the modify and will return to the prompt

COMMANDS

Commands are of the general form:

command parameters

All address values are in hex. A **q** can be used to terminate any command.

m a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents.The locations are displayed/modified one byte at a time.

mb a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one byte at a time.

mw a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one word at a time.

ml a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one longword at a time.

f a1 [a2] value The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fb a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fw a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this

occurs on a word basis. If a2 is not specified then only location a1 is changed.

fl a1 [a2] value The contents of memory locations a1 through a2 are filled with value. This occurs on a longword basis. If a2 is not specified then only location a1 is changed.

M a1 disp This command will cause the memory modify mode to be entered. Memory will be 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 This command will copy a block of memory starting at a1 through a2 and place at a3.

rm regname This command will allow the contents of memory mapped IOP based registers to be examined or modified. The contents of the register will be displayed. Entering a [return] will cause the next register to be displayed. Entering a hex value followed by a return
will change the contents of the register if it is a writeable register. Entering a ^ will cause the previous register to be displayed . Entering a **q** will exit this mode. The valid register names are:

ier	interrupt enable register
ierm	interrupt enable register (most significant word)
ierl	interrupt enable register (least significant word)
isr	interrupt status register
isrm	interrupt status register (most significant word)
isrl	interrupt status register (least significant word)
mcr	memory control register
pis	pbus interrupt channel
pic	pbus interrupt channel number
pblgm	pbus log register (most significant word)
pblgl	pbus log register (least significant word)
clg	cache log
earm	error address register (most significant word)
earl	error address register (least significant word)
trr	test results register
mbdr	multibus diagnostic register
misc	miscellaneous diagnostic register
pstat	parity status register
slotid	slot id register

rd regname This command will display the bits in the named register. Each bit position will be labeled with the function of the bit. Each bit can be individualy modified by moving the cursor over the bit and entering a 0 or a 1. The cursor will move to the right (towards the lsb) when a new bit value is entered or when the space bar is depressed. The cursor will move to the left when a DEL or BS key is pressed. The input is terminated when a RETURN , **q** , ^ or = is entered. A RETURN will update the register and then display the next register. A **q** will

exit this mode. Entering a ^ will update the register with the modified value then display the previous register. Entering a = will update the register with the modified value then redisplay the same register.

q[uit] This command will exit **ioputil.**

? This command will print out a list of valid commands.

NAME

map – display logical to physical mapping

SYNOPSIS

map a1 a2 ... an

DESCRIPTION

Map displays the segment descriptor register (SDR), the contents of the first and second level page table entries (PTE1, PTE2), and the physical address corresponding to the logical addresses **a1, a2, ...** and **an.** If the valid bit is not set for the SDR or a PTE, then question marks are printed for the remaining translation values.

NAME

margin – set C1 power supply and system clock margins

SYNOPSIS

margin [-n/u/l/x/v/g] [ps/ps1/ps2/ps3/clk/all] [-d -n/u clkccu

DESCRIPTION

Margin provides a means for the **C1** system clock and power supply margins to be set. Nominal (**-n**), upper (**-u**), lower (**-l**), and one-half nominal (**-x**) margins are available on the + 5 volt power supplies. They have the following effect on the power supplies and/or system clock:

Switch Resultant Voltage

-n	nominal voltage (typically +5 volts)
-l	lower margin (typically +4.75 volts)
-u	upper margin (typically +5.25 volts)

Resultant Clock Rate

-n	nominal frequency
-l	lower margin frequency
-u	upper margin frequency
-x	one-half nominal frequency

The **C1** power supplies and system clock may be margined individually, or in any combination. The following are accepted mnemonics:

ps1	power supply one (CPU power; + 5 volts)
ps2	power supply two (MAU, SPU, I/O power; +/- 5 volts, +/- 12 volts)
ps3	power supply three (optional I/O power; + 5 volts)
ps	all power supplies present
clk	system clock
all	all of the above

After setting the specified margin conditions, **margin** measures the resultant clock frequency and power supply voltages. Although only the + 5 volt power supplies can be **margin**ed, all the power supplies are monitored. If any clock or voltage exceeds the 10% tolerance, **margin** displays a diagnostic message naming the clock or voltage in error, the current reading, and the acceptable tolerance limits. If the **-v** option is given, all voltages and clocks are displayed regardless of tolerance conditions. If the **-g** option is specified, only those voltages or clocks out of tolerance are displayed. If **margin** is invoked with no arguments, the current **margin** conditions are displayed.

The -d option enables distructive features of margin and may be used to margin VIOP clocks. The clocks for the microprocessor section of a VIOP are seperate from the rest of the system. The mnemonic clkccu allows these clocks to be margined, however this margining requires the microprocessor section of the VIOP to be reset, and is therefore a distructive operation. The clkccu mnemonic is followed a slot number or the word "all" to specify which CCU's are to be affected by the command.

EXAMPLE

margin -l clk -n ps3 -u
margin -d -l clkccu all

RETURNS

Margin returns a zero if all power supplies and clocks are within tolerance, or non-zero if any are not within tolerance.

FILES

/mnt/bin/initall

SEE ALSO

initall(1D)

NAME

mm – display/modify main memory

SYNOPSIS

mm [-s]

DESCRIPTION

Mm is a utility to display and modify CONVEX-1 main memory and the population configuration map (PCM). When initiated with no options, **mm** saves the system clock state, halts the central processor, and initializes the MCU scan ring (tag parity errors cleared, EDC enabled, red LED off and green LED on). The initialized ring contents are saved and restored along with the initial system clock state upon normal termination of **mm** via the **q** command. After ring initialization is complete, **mm** displays a prompt of **mm:** and is ready for command input. When invoked with the *-s* (non-scan) option, **mm** proceeds to its internal prompting without the initialization sequence mentioned above. This allows the use of **mm** when the CPU is running CONVEX UNIX.

COMMANDS

Commands are of the general form:

command parameters

Unless noted otherwise, all numeric values are in hex. All address values represent physical addresses. <CNTRL-C> may be used to abort an operation and return to the **mm** prompt.

d a1 [a2] The contents of locations *a1* through *a2* are displayed in hex and ASCII.

d pcm The contents of the population configuration map are displayed. (This command is not available when the *-s* switch is used.)

e [on | off] Enable or disable error detection/correction (EDC) for the duration of **mm**. Default is EDC on. If no argument is entered, the current state of the EDC switch is displayed. (This command is not available when the *-s* switch is used.)

f a1 a2 patt The hexadecimal pattern *patt* is repetitively copied into locations *a1* through *a2*.

h Display command summary.

i Memory is sized, and the PCM is set accordingly. All of main memory is cleared, and error detection/correction (EDC) is enabled. Hard and soft interrupts are enabled on the memory control unit (MCU). (This command is not available when the *-s* switch is used.)

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:

<cr> Increment address
- Decrement address
= Display same address
g Go to address specified by new_data. Do not modify old_data.
q Return to mm prompt

m pcm The PCM modification mode is entered at block 0. The PCM modification

mode uses the same operation codes as the memory modification mode. For the **g** command, the address is specified in the form *m/b*, where *m* is the MAU number and *b* is the decimal PCM block number on the specified MAU. The address may also be specified in the form *b* or *mb* where *m* and *b* are only valid for values 0 through 7. (This command is not available when the *-s* switch is used.)

q Restore the system clock state and MCU scan ring, then exit **mm**.

Q Restore the system clock state and exit **mm**. The MCU scan ring is not restored. (This command is not available when the *-s* switch is used.)

sy params This command allows symbols to be defined, displayed, or deleted. Symbol names may be up to 15 characters long. The first character of the symbol must be a period. Symbol definitions are stored in the file *mm#sym* in the current directory. The following forms of the symbol command are available:

sy	Display all symbols
sy .name	Display value of *.name*
sy .name value	Assign a value to *.name*
sy -.name	Delete *.name*
sy --	Delete all symbols

FILES

mm#sym

BUGS

After power up, the memory system must be reset (see **sysreset(1D)**) before **mm** can access main memory. Additionally, the PCM and main memory must be initialized by **mm** (or **mminit(1D)**) before other utilities can access main memory.

NAME

mm_sniff – main memory sniffer

SYNOPSIS

mm_sniff [*-r nn* | *-t nn*]

DESCRIPTION

The **mm_sniff** program reads all main memory within a specified amount of time. The intention is to detect locations with a single-bit error. Once detected, **errintd** 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 Central Processing Unit (CPU).

The memory sniffer always reads main memory in four-page groups (e.g., 16 Kbytes, 1/64 the size of one Population Configuration Map (PCM) entry). Upon invocation, based on the sniff rate in Mbytes/day, **mm_sniff** calculates the number of seconds to sleep between each read. In the event the calculated sleep time is less than 15 sec, the value is forced to 15 sec. This time and the time required to sniff the entire memory system are reported.

The following options are interpreted by **mm_sniff**:

–r nn Set sniff rate to *nn* Mbytes/day. The default is 32 Mbytes/day.

–t nn Set sniff sleep time to *nn* sec. This option overrides the **–r** option.

SEE ALSO

errintd(1D)
softlog(5D)

NAME

mminit – main memory initialization

SYNOPSIS

mminit [-m] [-p n] [-s] [-c]

DESCRIPTION

Mminit is used to initialize main memory after system power up. Normally, **mminit** sizes main memory, initializes the population configuration map (PCM), clears all locations in main memory, and displays a table of the memory blocks found to be present. By default, **mminit** searches the main memory address space for working memory blocks to initialize the PCM. For each physical memory block that responds, a corresponding bit in the PCM is set. 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, **mminit** returns a 0.

The following options are supported:

-c If this option is specified, the PCM is initialized based on the part number that has been programmed into the COP chip on each MAU.

-m Initialize main memory only. It is assumed that main memory has already been sized and the PCM initialized.

-p n Set the memory initialization pattern to the long word hex pattern specified by **n.** The default long word pattern is 0.

-s Use the Service Processor Unit (SPU) to perform the initialization via the PBUS. No vector operations are performed, and the central processor is not involved in the initialization. This mode of initialization is significantly slower than the default vectorized initialization mode.

DIAGNOSTICS

Error messages are printed describing memory areas that do not respond. If an error condition exists on systems that support variable memory interleaves, memory interleave is set to minimum. (See **interleave(1D)**).

FILES

/mnt/bin/initall

SEE ALSO

interleave(1D)

NAME

mmld – load an a.out file into C1 main memory

SYNOPSIS

mmld [-l] **file** [**offset**]

DESCRIPTION

Mmld calls **jpload(3D)** to load a C1 **a.out** file into main memory. **File** specifies the name of the file to be loaded. The file may have a magic number of either 507 or 513 and is loaded as a coreimage file. Unless **-l** is specified, all addressing is physical. The load address of the file is equal to the file origin plus the hex value specified by **offset.**

If **-l** is specified, then logical addressing is used during loading. The segment descriptor registers (SDR's) and page table entries (PTE's) are accessed to determine the appropriate logical to physical address mapping. All PTE's must be memory resident.

Mmld initializes the second level page table access bits as show below. The first level page table access bits are not initialized.

File Segment	Magic 507	Magic 513
Text	rd wr ex rs	rd wr ex rs
Data	rd wr ex rs	rd wr ex rs
Bss	rd wr ex rs	rd wr ex rs

SEE ALSO

SPU Unix: **jpload(3D)**
HOST Unix: */usr/include/a.out.h*

NAME

pup – power-up bit value read/write

SYNOPSIS

pup [**on** | **off**]

DESCRIPTION

The **pup** utility reads/sets/clears the power-up bit on the Service Processor Unit (SPU). **Pup** returns the previous value of the power-up bit if a parameter is specified, or the current value of the power-up bit (**0** or **1**) if no parameter is specified. Additionally, if no parameter is specified, the power-up bit remains unchanged. If any error occurs, **-1** is returned.

NAME

ringrev_chk - check for CPU scan ring revision changes

SYNOPSIS

ringrev_chk

DESCRIPTION

Ringrev_chk is used on power up to check if the CONVEX-1 CPU scan ring revisions have changed since the last power-up. **Ringrev_chk** performs its check using the data base files, */mnt/usr/scn/cop.out* and */mnt/usr/scn/cop.new*. The board revisions checked are: asu, atu, ipu, mcu, pcu, vcu, vpu0, vpu1, and ccu1,2,3,4,5,6,7 . A return code of zero indicates the scan ring revisions agree. If there are scan ring revision changes in the *cop.new* file that differ in any way from the *cop.out* file, then a return code of 1 is issued. Below is a complete listing of all return codes:

0= scan ring revisions agree
1= scan ring revisions do not agree
2= different board names are listed in *cop.out* than in *cop.new*
3= could not find *cop.out* file
4= could not find *cop.new* file
5= other error

SEE ALSO

cop(1D)
diaginit(1D)

FILES

/mnt/usr/scn/cop.out
/mnt/usr/scn/cop.new

NAME

scan – C1 interactive scan facility

SYNOPSIS

scan file

DESCRIPTION

The **scan** utility is an interactive software tool which allows a user to control and observe the internal states of individual boards of the **C1** central processor, and I/O processors.

Mnemonics (symbolic names) have been defined for the various bit fields within each scan ring. Synonyms have been defined for values within these fields, such as "on", "off", "enabled", "disabled", etc. These mnemonics and synonyms may (and are encouraged to) be used by the user when refering to fields and their values. In addition, various screens have been defined for each board. These screens depict the state of the scan ring in a format that shows all necessary scan data concerning a certain board function.

All of these mnemonic definitions and scan ring structure definitions are expressed in a text file, called the scan definitions file. To avoid parsing this file every time the scan utility is invoked, a separate compiler, **scanc** has been developed to parse and translate the textual description into a binary file. This compilation has been performed and should not be necessary in the diagnostic environment. The output file of the compiler is listed in the **scan** command line as the *file* argument, and should end with the suffix *.sco*

The scan utility, itself, is designed to work interactively through either a line oriented device, such as a traditional teletype; or with a screen oriented device supporting random cursor movement. A line oriented command set exists which supports the entire functionality of the scan utility, and relies only on the capabilities of a teletype. However, there is also a screen edit mode, in which the user may peruse and edit a predefined screen of displayed fields. This ability drastically reduces the number of keystrokes necessary to interrogate and modify ring data by generating equivalent line oriented commands for the user. The functionality of the screen editor is only a subset of the line oriented command set, but the user may enter and leave the edit mode at any time from the line oriented mode.

COMMAND SUMMARY

name	**alias**	**meaning**
all	a	list all ring values
allbits	ab	list ring bit fields
bits	b	list single bit field
cgr	C	display cgr register
clock	c	apply clock(s)
edit	e	interactive edit
editl	el	interactive edit
esr	E	display esr register
execute	x	execute file
executel	xl	execute file
get	g	get field value
help	?	help
iupdate	iu	set update flag
loadram	lr	apply load ram pulse
loadscan	ls	apply load scan pulse
log	l	create log file
logl	ll	create log file

put	p	put field value
putlog	pl	put string in log file
putlogl	pll	put string in log file
read	R	read ring(s)
reset	re	reset subsystem
run	r	run board(s)
screens	sc	list screen names
sh	!	execute shell
snapshot	sn	log ring values
snapshotl	snl	log ring values
verify	v	set verify flag
write	W	write ring(s)

for more information type:

help <command name>

COMMAND EXPLANATIONS

Following are descriptions of each of the commands listed in the command summary. This is essentially the same information programmed into the self help facility.

help (?) -- display help screen

This command displays helpful information. If no parameters appear, then the command summary is displayed. If a command name follows "help", then a help page for that command is displayed.

usage: help [<command name>]

examples: help
help snapshot

get (g) -- get field value

Displays the specified field value in hexadecimal and as a synonym if one is defined.

The value of the field is taken from the local copy of the ring data. This command never results in a hardware read.

usage: get <field name>

example: get mcu:red-led

put (p) -- put field value

Places the specified value in the specified field. An optional format may be specified following the "/". The default format is hexadecimal.

If the "iupdate" flag is "on", the "put" causes immediate write to the hardware scan ring. If it is "off", only the local copy of the scan ring is updated, and the hardware write does not occur until a "write" or "iupdate on" command is executed.

usage: put <field name> <hex value>
put <field name> /b <binary value>
put <field name> /s <synonym value>
put <field name> /h <synonym value>

example: put mcu:red-led /s on

see also: iupdate (iu), verify (v), write (W), edit (e)

verify (v) -- set verify flag on/off.

Indicates if read/compare verifications are to be done after writes to the hardware scan rings. If the flag is "on", a read and compare is performed after each scan ring write. If a discrepancy is detected, an error message is displayed, including a hexadecimal number indicating the bits which differed between the written value and the read value. The verify flag is initialized to "on".

usage: verify [on | off]

If no parameters, then "on" is assumed.

examples: verify
verify on
verify off

see also: iupdate (iu), write (W), put (p), edit (e)

iupdate (iu) -- set immediate update flag.

Indicates whether scan writes are to be performed each time a scan ring field is updated via a "put" command. If the immediate update flag is "off", then only the local copy of the scan ring is updated when the "put" is executed, and the actual hardware is not affected. If the flag is "on", then any "put" command will cause the scan ring to be written immediately with the new field value. Executing an "iupdate on" also causes immediate writes of any rings which have been modified since last read or written.

usage: iupdate [on | off]

If no parameters, then "on" is assumed.

examples: iupdate off
iupdate on
iupdate

see also: put (p), verify (v), write (w), edit (e)

read (R) -- read hardware scan ring(s) into local copy.

This command causes the indicated (or implied) hardware scan rings to be read into SCAN's local

scan ring data copies.

SCAN maintains a local copy of scan ring data for each defined scan ring. These copies are initialized when SCAN starts up via an implied "read" command, and are used throughout the SCAN session for "get" and "put" commands. If the "verify" flag is "on", then an implied "read" will also be done after each scan ring hardware write for verification. These are the only two cases where "read"s are executed, other than the explicit execution of the "read" command. Implicit reads may be avoided by setting "verify" to "off".

usage: read [<ring name> ...]

If no parameters, then all defined rings are assumed.

examples:
```
read ipu mcu    # reads only ipu and mcu
read            # reads all defined scan rings
```

see also: get (g), put (p), verify (v), write (w), edit (e)

write (W) -- write hardware scan ring(s).

This command causes the indicated (or implied) hardware scan rings to be written from SCAN's local scan ring data copies.

SCAN maintains a local copy of scan ring data for each defined scan ring. A "put" command causes an immediate write to the hardware scan ring if the "iupdate" flag is on. If "iupdate" is off, then "put" only causes the local copy of the scan ring data to be updated, and no hardware write takes place. When "iupdate" is "off", the only time hardware writes occur is when a "write" command is explicitly executed.

If the "verify" flag is "on", then a read/compare operation is performed after each implicit or explicit hardware write. When "verify" is off, then no implicit reads occur.

usage: write [<ring name> ...]

If no parameters, then all defined rings are assumed.

examples:
```
write ipu mcu   # writes only ipu and mcu
write           # writes all defined scan rings
```

see also: iupdate (iu), verify (v), read (R), get (g), put (p), edit (e)

reset (re) -- reset indicated subsystem.

This command causes the indicated subsystem of the CONVEX computer to be reset. The reset is performed by applying 16 clocks to the subsystem boards with the appropriate bit set in the system reset register (SRR).

usage: reset [jp | hsc | mem | mult]

If no parameters, then all subsystems are reset.

```
examples:   reset jp      # resets only job processor boards
            reset         # resets all subsystems
```

log (l), logl (ll) -- establish/terminate command log file.

These commands establish a file in which commands will be written as they are executed. The log file is maintained in a format that can be executed later via the "execute (x)" command to recreate the SCAN session.

Certain commands do not get logged, because their execution at a later time would not reproduce the original effect. These commands are: "log (l)", "execute (x)", "edit (e)", and "snapshot (sn)". However, there are alternate forms for these commands which will get logged, and executed if log file is ever executed. These are respectively: "logl (ll)", "executel (xl)", "editl (el)", and "snapshotl (snl)". Initially no log file is established, and a "log (l)" or "logl (ll)" command with no parameters will close the current log file (if any), and establish no logging of commands.

```
usage:      log [ <file name> ]

examples:   log setup.log   # establishes "setup.log" as log file
            log             # establishes "no log file".

see also:   lognotes, snapshot (sn), execute (x), edit (e)
```

execute (x), executel (x) -- execute file of commands.

These commands cause the command interpreter to begin executing commands from a specified file. The command interpreter continues executing commands from the file until end of file is reached, at which time it returns to its previous source for commands. One command file may have an execute command for another command file. The command file may be a previous log file or an edited file.

"execute (x)" itself is not logged, "executel (xl)" is.

```
usage:      execute <file name> [ e ]
```

If "e" is specified, then the commands are echoed to the terminal as they are executed. Otherwise the commands execute silently, except for any displays they themselves may cause.

```
examples:   execute setup.log     # execute silently
            x snap.log e          # execute and echo commands

see also:   log (l), snappshot (sn), edit (e)
```

edit (e), editl (el) -- edit screen interactively.

These commands invoke the interactive screen editor. ("edit (e)" is not logged, "editl (el)" is logged.) The screen editor simplifies the process of interrogating and updating scan ring values,

by allowing the user to peruse and edit fields of a pre-defined screen. The editor acts as a "command generator" to the rest of the SCAN utility. For example, when a displayed field is modified, the editor generates a "put" command, displays it at the bottom of the screen, and passes the command to the SCAN command line interpreter for execution. Therefore, the functional capabilities of the editor compose a subset of the functional capabilities of the line-oriented command set.

The editor generates the following commands: "put (p)", "read (R)", and "write (W)". If a log file is established, these commands are logged as they are generated, and therefore, the effects of an edit session may be reproduced later by executing the log file. Use ^? (be sure and shift) for help from editor.

usage: edit <screen name>

examples: edit state-screen

see also: log (l), execute (x), put (p), read (R), write (W)

snapshot (sn), snapshotl (snl) -- put snapshot in log file.

These commands cause "put" commands to be generated and logged in the current log file, making a snapshot of the specified (or implied) ring values. A "put" statement is generated for each field of each specified ring with the current value of that field. Executing this log file at a later time will result in the scan rings being restored to their values at the time the snapshot was made.

The "snapshot (sn)" command itself is not logged, "snapshotl (snl)" is logged.

usage: snapshot [<ring name> ...]

If no parameters, then all defined rings are snapshot.

examples: snapshot mcu ipu
snapshot

see also: log (l), execute (x), put (p)

putlog (pl), putlogl (pll) -- put string in log file.

These commands cause all operands of the command to be placed into the current log file. If the command is executed from the keyboard, no parameter substitution occurs; the operands are placed into the log file as is. If the command is executed from a file (via "x" command), parameter and field substitutions occur before the operands are placed into the log file. This command is very handy for generating commands to be executed later.

The "putlog (pl)" command itself is not logged, "putlogl (pll)" is logged.

usage: putlog <parameters>

examples: putlog print The value of $1 is ${$1}.

```
putlog putlog put $1 ${$1} # snapshot field $1
```

see also: log (l), execute (x), put (p)

loadram (lr) -- apply load ram pulse to selected boards.

This command cause a single load ram pulse to be generated for a specified set of boards. The DMODE line is asserted as an option.

usage: loadram [d] [<ring name> ...]

If "d" is present as the first parameter, then DMODE is asserted during the load ram pulse. If no rings are specified, then all defined rings are selected.

```
examples:   loadram d ipu mcu    # load ram with DMODE to ipu and mcu
            loadram                    # loadram to all defined rings
            loadram ipu          # loadram to ipu only
```

loadscan (ls) -- apply load scan pulse to selected boards.

This command cause a single load scan pulse to be generated for a specified set of boards. The DMODE line is asserted as an option.

usage: loadscan [d] [<ring name> ...]

If "d" is present as the first parameter, then DMODE is asserted during the load scan pulse. If no rings are specified, then all defined rings are selected.

```
examples:   loadscan d ipu mcu   # load ram with DMODE to ipu and mcu
            loadscan                   # loadram to all defined rings
            loadscan ipu         # loadram to ipu only
```

clock (c) -- apply specified number of clocks.

This command causes a specified number of clocks to the generated for a specified set of boards. The DMODE line is asserted as an option. The ^C key may be used to interrupt the execution of a lengthy clock command.

usage: clock [d] [<number>] [<ring name> ...]

If "d" is present as the first parameter, then DMODE is asserted during the load ram pulse. If the number is not specified, then a single clock is generated. If no rings are specified, then all defined rings are selected.

```
examples:   clock d              # 1 clock w/ DMODE all defined rings
            c                    # single clock to all defined rings
            clock 1000000 ipu    # 1000000 clocks to ipu only
```

run (r) -- supply continuous clocks.

This command causes a specified set of boards to be placed into a run state. The ^C key (DELETE on HOST) may be used to interrupt the execution of a run command.

usage: run [<ring name> ...]

If no rings are specified, then all rings are selected.

examples:
```
run ipu mcu     # run ipu and mcu
run             # run all boards
```

cgr (C) -- display/modify clock gating register.

This command displays the CGR or updates it.

usage: cgr [<hex number>]

If the number is not present, then the current value of the CGR is displayed. If the number is present, it is placed in the CGR.

examples:
```
cgr 00010000    # set CGR
cgr             # display CGR
```

esr (E) -- display error source register.

This command displays the ESR.

usage: esr

examples: esr

sh (!) -- execute shell.

This command causes a shell to be executed. A ^D terminates the shell and reenters SCAN.

usage: sh

examples: sh

bits (b) -- display bit field descriptors.

This command displays the bit field descriptor for the specified field.

usage: bits <field name>

examples: bits mcu:red-led

allbits (ab) -- display all bit field descriptors.

This command displays the bit field descriptors for all fields of the specified ring.

usage: allbits <ring name>

examples: allbits mcu

all (a) -- display all values.

This command displays the values of all fields in the specified ring.

usage: all <ring name>

examples: all mcu

screens (sc) -- display names of all defined screens.

This command displays the names of all defined screens.

usage: screens

examples: screens

SEE ALSO

scanc(1D)

NAME

scanc – C1 scan definition file compiler

SYNOPSIS

scanc <infile.scd >ofile.sco
scanc filename

DESCRIPTION

The scan compiler, **scanc,** translates the scan definitions file into a binary file, which is used as input to the interactive scan utility, **scan(1D).**

The scan compiler reads the definitions from its standard input file, *infile.scd* and if no errors occur, places the binary into the standard output file (filenameFI.sco). If errors occur, a message is written to the standard error file (filename.*sce*), and the standard output file (filename.*sco*) contains a copy of the definitions file with error messages placed appropriately.

SCAN DEFINITIONS FILE

The scan definitions file is a text file containing four types of specifications:

- O Synonym List Specifications (synlist)
- O Format Specifications (format)
- O Ring Specifications (ring)
- O Screen Specifications (screen)

Together, these specifications identify the following:

- O bit locations of fields withing scan rings.
- O symbolic names of fields.
- O synonyms for field values.
- O screen formats for interactive edit.

The file is created and modified easily with any text editor, and must be compiled with the scan compiler, **scanc** before using the interactive scan utility.

The definitions are expressed in a simple, block structured syntax, which looks very similar to "struct" definitions in the "C" programming language. The compiler is single pass, hence, forward references are not allowed. This presents no problem, however, as long as the specifications occur in the following order:

1. All Synonym List Specifications.
2. All Format Specifications.
3. All Ring Specifications.
4. All Screen Specifications.

To facilitate the explanation of this syntax, upper case words will be used to indicate non-terminals. That is, they represent a "type" of expression that may appear in that place. ("DECIMAL_NUMBER" means any valid decimal number.) Except for two cases, all other characters represent themselves, and must appear in the definitions file in that place. ("format" means "format", "<" means "<" .) The two exceptions are: square brackets ("[ ]") indicate that the enclosed expression is optional, and three dots ("...") indicate that the preceding line of the description may be repeated zero or more times.

Following is a description of the definitions syntax. This is an informal description of the syntax, not a rigorous definition. It is assumed that this type of description is most useful in this case.

The syntax is free-format. There are no indentation rules, or dependencies on end of line. The end of line (or "newline" character) is treated as white space, along with "tab" and "blank". Comments may occur anywhere white space is allowed, and must be enclosed in "/*" and "*/" (just like C and PLI). (No space allowed between the "/" and "*".)

```
/* This is a comment */

/*
 * This is also a comment ...
*/

/  * This will cause a compiler error *  /
```

White space may occur anywhere between tokens.

The non-terminal "SYMBOL" indicates a user-defined symbol. A SYMBOL is a sequence of characters beginning with a non-digit, and containing any characters except the following:

<	left angle bracket
>	right angle bracket
{	left curly bracket
}	right curly bracket
.	dot
:	colon
;	semicolon
=	equal
/	slash
	white space (blank, newline, tab, etc.)

The occurrence of any of these characters terminates the symbol, and possibly begins the next token. Therefore, it is legal to have a user defined symbol immediately followed by a ";", if the semicolon is allowed after the symbol. (i.e. white space is optional here.) Following are examples of valid symbols:

```
cache_state

x_bus_source

wrd_code(1-0)*          /* this is a single valid symbol */
```

Following are examples of INVALID symbols:

```
cash state          /* WRONG! Actually two symbols */

source/dest         /* WRONG! "/" not allowed in symbol */

wrd_code(1..0)*     /* WRONG! "." not allowed in symbol */
```

The non-terminal "DECIMAL_NUMBER" represents any contiguous string of decimal digits, optionally preceded by "-".

Synonym List Specification

Synonyms for numeric values are defined in "synonym lists". Each synonym list has a SYMBOL for a name, and a list of SYMBOLS with associated numeric values represented as SCALD_NUMBERS. The synonym list specification is started with the word "synlist". Following is the template for a synonym list specification:

```
synlist SYMBOL
 {
  SYMBOL = SCALD_NUMBER ;
  ...
 };
```

A SCALD_NUMBER is a number represented in SCALD format. The number may either be a simple decimal number, or may have a radix specification of 2 to 16 (expressed in decimal), followed by #, followed by the number in the specified radix. (The letters "a" .. "f" are used to represent digit values of 10 .. 15.) Following are some examples of valid SCALD numbers:

```
36              /* decimal 36 */

2#100010        /* binary 100 010 */

16#ffff         /* hexadecimal ffff */
```

Following is an example of a synonym list:

```
synlist fp_status
 {
  normal   = 0;
  overflow = 2#001;
  undrflow = 2#010;
  divzero  = 2#100;
 };
```

The symbols within each synonym list must be unique with respect to other symbols in the SAME synonym list, but may be used again in other synonym lists, and have different values there. This allows the name of a synonym list to establish a context for the interpretation of field

values. For example, consider the following two synonym lists:

```
synlist on_off
   {
     on = 1;
     off = 0;
   }; /* END synlist on_off ...*/

synlist on_off* /* inverted on/off logic */
   {
     on = 0;
     off = 1;
   }; /* END synlist on_off* ... */
```

The synonym list "on_off" can be used in the specification of fields (discussed shortly) where "1" indicates an "on" condition. Other fields may interpret "0" to be "on", and therefore the "on_off*" synonym list may be specified for those fields. Many times, this capability relieves the designer from remembering where logic is inverted.

Ring Format Specification

The ring format specification (or "format" specification) defines fields within rings. This definition includes:

- O the symbolic name of the field.
- O the locations of the bits composing the field.
- O the associated synonym list (optional).

A format definition begins with the word "format", followed by a SYMBOL for the format's name, followed by one or more field definitions, each terminated by a semicolon. Following is the template of a format specification:

```
format SYMBOL
   {
     FIELD_DEFINITION
     ... /* as many as you want */
   };
```

The FIELD_DEFINITION begins with a SYMBOL which represents the name of the field, optionally followed by a BIT_FIELD_DESCRIPTOR, optionally followed by a ":" and synonym list name, and terminated with a ";". Following is the template of a FIELD_DEFINITION:

```
SYMBOL [ BIT_FIELD_DESCRIPTOR ] [ : SYMBOL ] ;
```

The BIT_FIELD_DESCRIPTOR indicates the bits in the scan ring that comprise the field. The bits need not be contiguous, and can be defined to exist in more than one field. (This is sometimes done to show hierarchy.) The format for the BIT_FIELD_DESCRIPTOR is general enough to allow a contiguous range of bits, with either end being most significant; or a random list of bits.

The syntax for this specification is the same as is used in the SCALD language on VALID for <subscript> fields, with some extensions to ease modification of the scan ring formats. The BIT_FIELD_DESCRIPTOR may be any of the following forms:

```
< BIT >
< BIT .. BIT >
< BIT .. BIT : STEP >
< BIT : WIDTH >
< : WIDTH >          ***** extension *****
< BIT : WIDTH : STEP >
< : WIDTH : STEP >    ***** extension *****
< BIT_LIST >
```

BIT and STEP are nonnegative decimal numbers, WIDTH is DECIMAL_NUMBER (possibly negative). A negative WIDTH implies reversed significance. BIT_LIST is a list of nonnegative bit numbers separated by commas.

The extensions have an implied bit number adjacent to the previous field. If no BIT_FIELD_DESCRIPTOR exists in a field definition, the field is assumed to be one bit wide and adjacent to the previous field. Although formats which have implied starting bits allow the designer to insert fields into the format specification without editing all of the bit numbers that changed, the practice is not encouraged.

Following is an example of a complete format definition:

```
format ipu_format
 {
  state < 407 .. 404 > : ipu_state;
  red_led: < 104 >: on_off;
  green_led*: on_off*;    /* adjacent to red_led */
 }; /* END format ipu_format ... */
```

The scan utility also determines ring size from the format descriptions, and therefore, the top bit of the scan ring must be referenced, even if never used.

Ring Specification

Each physical scan ring to be interrogated or modified during a scan session must be defined with a "ring specification". A ring specification identifies:

- the symbolic name of the ring.
- the "bit number" (logical address) of the ring in the CMR.
- the symbolic name of the format associated with the ring.

The reason that the "format" and "ring" definition are separated is because there is at least one case where two boards have the same format, but different bit numbers.

The ring specification begins with the word "ring", followed by "<", the bit number, and ">", followed by ":" and the name of format, and terminated with ";". Following is the template for a ring specification:

```
ring SYMBOL < BIT_NUMBER > : SYMBOL ;
```

BIT_NUMBER is a DECIMAL_NUMBER in the range 0..31 . Following is an example of a ring specification:

```
ring ipu < 20 > : ipu_format;
```

Screen Specification

Any number of screens may be defined to be edited with the interactive screen editor. Each screen requires a "screen specification" which identifies:

- the symbolic name of the screen.
- the fields to be displayed on the screen.
- (optionally) the row and column of the screen at which each field is to be displayed.
- the format(s) used to display each field.

Each screen definition begins with the word "screen", followed by the symbolic name of the symbol, and a list of display field descriptions. Any one screen may contain fields from several different rings. Therefore, the designers may want to define screens with logically associated fields from different boards, so that the interaction among the boards can be monitored on a single screen with respect to a logical function. Following is the template for a screen specification:

```
screen SYMBOL
 {
  SYMBOL:SYMBOL [ < ROW , [ COL ] > ] : FORMATS ;
  ...
 }; /* END screen SYMBOL ... */
```

The ring fields are identified by the ring name, followed by a ":", followed by the field name (in the associated format specification). An optional row/column specification may follow which specifies where on the screen the field is to be displayed. If this is omitted, the compiler will automatically select a convenient place to put it. If present, the row must be specified and must be in the range 1..20 . The column may be omitted, indicating that anywhere on the specified row is ok.

The FORMATS specification is a string consisting of any combination of "s", "h", and "b". "s" indicates that the field value is to be displayed in symbolic form (according to its synonym list specification). "h" indicates hexadecimal, and "b" indicates binary. A field may be displayed in more than one format. Following is an example of a screen specification:

```
screen general
 {
  ipu:red_led : sb; /* display anywhere in symbolic and binary */
  ipu:green_led < 5, > : sb; /* display on row 5 */
  ipu:state < 10, 10 > : h; /* display in hexadecimal */
 }; /* END screen general ... */
```

SEE ALSO

scan(1D)

NAME

scn_ring – interactive scan ring read/write/check utility

SYNOPSIS

scn_ring

DESCRIPTION

Scn_ring is a utility that allows the user to interactively read, write, or check data patterns to/from a CONVEX-1 scan ring. **Scn_ring** prompts the user for the information needed to perform the requested operation. The *operation types* are:

Check a scan ring for integrity to write/read a pattern.

Write a data pattern to a scan ring.

Read the data from a scan ring.

Exit to the shell.

After the user selects the *operation type* desired, **scn_ring** prompts the user with the options available for the operation and checks for valid responses to each of the prompts. The prompts are used by **scn_ring** to obtain the following information:

Board type - options included for all slots.

Write direction - allows pattern to be written to left or right.

Read direction - allows pattern to be read from left or right.

Pattern type - options allow a pattern of a *one* on a background of *zeros* or a *zero* on a background of *ones*.

Enter scan bit location to receive [1/0] - where to put the pattern bit in the data pattern.

DIAGNOSTICS

Scn_ring allows only valid responses to the option list.

NAME

scnlink – intermediate scan ring definition file linker

SYNOPSIS

scnlink [**-m**] [**file**]

DESCRIPTION

Scnlink is an SPU utility which generates the intermediate scan ring definition file, */mnt/usr/scn/scn_rings*, used by all SPU routines which access the scan rings. Pin assignments for each scan ring are defined by a set of files in the */mnt/usr/scn* directory. These files are of the form (boardname)*_rev*(number), such as *ipu_rev3*. When **scnlink** is executed on the SPU, it reads a file generated by the **cop** utility which specifies the actual boards and revisions installed in the system. This file is either specified in the invocation of **scnlink**, or */mnt/usr/scn/cop.out* is used if no file is specified.

If the **-m** flag is specified, **scnlink** builds the file */mnt/usr/scn/scn_rings2*, which defines the board characteristics for slots which do not have scan rings (SPU and MAU's). If the optional file is not specified in this case, the file */mnt/usr/scn/cop.mem* is used.

The individual scan ring description files are generated on a host system by the **usr/scn** utility.

SEE ALSO

scn(3D)
cop(1D)

FILES

SPU:

/mnt/usr/scn/scn_rings
/mnt/usr/scn/scn_rings2
/mnt/usr/scn/cop.out
/mnt/usr/scn/cop.mem
/mnt/usr/scn/(boardname)*_rev*(number)

NAME

security_clear – memory and cache purge

SYNOPSIS

security_clear [**-i**] [**-no_hia**] [**-e nnn**] [**-E nnn**] [**-f pattfile**] [**cpu**] [**ccus**] [**spu**] [**all**]

DESCRIPTION

The utility **security_clear** writes and verifies patterns to Central Processor Unit (CPU) caches, CPU main memory, Channel Control Unit (CCU) caches, CCU memory, High-Speed Parallel (HSP) buffers, HSP Interface Adapter (HIA) buffers, and Service Processor Unit (SPU) memory. It outputs the processor and memory range being purged as well as outputting when the write starts and ends and when the verify starts and ends. Any miscompares are displayed. The processors include the CPU, SPU, Input/Output Processor (IOP), VMEbus Input/Output Processor (VIOP), and HSP with associated caches.

The following options are supported:

-i When specified, **security_clear** stops after each verify to allow inspection of the memory. At inspection time, specify a range of addresses to dump, specify a single address that displays one line of data, enter **q** to quit inspection of the just-cleared area, or enter **cancel** to cancel any further prompts for inspection. If only one address is entered, then one line is displayed which permits choosing return to dump another line, typing a new address where the next displayed line starts, or entering a **q** to return to the **Inspect:** prompt.
Default: All memories purged with no interactive inspection.

-no_hia
This option prevents the CCU purge logic from attempting to purge HIA buffers. This may be necessary if something other than an HIA is attached to an HSP.
Default: A test is made to confirm an HIA exists and then the HIA buffers are purged.

-e nnn
If this option is specified, then if more than nnn errors occur during a single pattern, the pattern is skipped and the next pattern is started.
Default: 2,147,483,647

-E nnn
Use this option to terminate **security_clear** due to an excessive number of errors occurring during a single pattern. Termination occurs when the number of errors exceeds nnn.
Default: 2,147,483,647

-f pattfile
Defines the name of a file that contains separate patterns to be used with each processor.
Default: Searches for **purge_patterns** in the current directory first and then in */mnt/bin/lib*.

[**cpu**] [**ccus**] [**spu**] [**all**]
Specifies which processors will be purged. Implicitly specify **all** by not specifying any of the above processors.
Default: **all**

The above parameters can be specified in any order.

It is possible to specify a different pattern file than *purge_patterns* by using the **-f** option in the command parameters.

The pattern file specifies patterns for each processor. First specify a processor (**ccu**, **cpu**, or **spu**), followed by patterns for that processor. Each pattern must be 8 bytes long (16 hexadecimal digits including leading zeroes) and must be separated from other patterns for a given processor with white space (spaces, tabs, or newlines). Up to 64 patterns can be

specified per processor. If **random** is entered as one or more of the patterns, then an 8-byte random pattern is generated and used. The same 8 bytes will be written repetitively to all the area being purged and then will be verified.

A comment can be placed anywhere in the file by using a **#**. From that point to the end of the line is ignored.

Example of the **purge_patterns** file:

```
# This is a sample purge_patterns file
cpu 0000000000000000 ffffffffffffffff random    # three patterns for cpu
ccu aaaaaaaaaaaaaaaa                            # one pattern for ccus
spu random                                      # two patterns for spu
   2222222222222222
```

When **random** is specified, an 8-byte number is generated from current time by calling the function *srand(time(0))* to seed the random number generator and then by calling *rand()* eight times to get eight 1-byte values from which the 8-byte pattern is constructed. Each specification of **random** in the pattern file will result in eight new calls to rand() to generate a new 8-byte random number.

FILES

/mnt/bin/security_clear
/mnt/bin/lib/security_clear/purge_cpu
/mnt/bin/lib/security_clear/purge_iop
/mnt/bin/lib/security_clear/purge_viop
/mnt/bin/lib/security_clear/purge_hsp
/mnt/bin/lib/security_clear/mm_purge_ccu
/mnt/bin/lib/security_clear/mm_purge_spu
/mnt/bin/lib/security_clear/purge_patterns

SEE ALSO

rand(3)
srand(3)

NAME

sfpread – read/modify the SPU front panel switches

SYNOPSIS

sfpread [**-i**] [[**-v**] **field-name**]

DESCRIPTION

Sfpread is used to read and modify the contents of the SPU front panel. This utility operates in two different modes. If **field-name** is specified (with or without the **-v** option), a value corresponding to the current setting for that field is returned to the shell. If **-i** (interactive) is specified, **sfpread** emulates the front panel program **spu1000** and thus allows the front panel switches to be modified on a running system.

If the **-v** (verbose) option is supplied in addition to a field-name, sfpread prints the specified field name and its value in addition to returning that value to the shell.

DIAGNOSTICS

If the specified field-name does not exist or if any other error condition is found, **sfpread** exits with exit status -1. However, if invoked with the **-v** option, **sfpread** always exits with status zero.

FILES

/mnt/bin/sfpread

NAME

spuutil – spu register/memory utility

SYNOPSIS

spuutil

DESCRIPTION

Spuutil is a utility to display and modify SPU memory locations. When the modify mode is entered the displayed value can be changed by entering the new hex value and hitting RETURN. If no value is entered the next memory location will be displayed. A **q** will terminate the modify and will return to the prompt

COMMANDS

Commands are of the general form:

command parameters

All address values are in hex. A **q** can be used to terminate any command.

m a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents.The locations are displayed/modified one byte at a time.

mb a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one byte at a time.

mw a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one word at a time.

ml a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one longword at a time.

f a1 [a2] value The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fb a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fw a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this

occurs on a word basis. If a2 is not specified then only location a1 is changed.

fl a1 [a2] value The contents of memory locations a1 through a2 are filled with value. This occurs on a longword basis. If a2 is not specified then only location a1 is changed.

M a1 disp This command will cause the memory modify mode to be entered. Memory will be 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 This command will copy a block of memory starting at a1 through a2 and place at a3.

tg a1 cnt val This command will write val to a1 then wait for a period of time proportional to cnt; then write out the exclusive-or of val out to a1. This command will continue until interrupted by cntl-c or cntl-?.

rm regname This command will allow the contents of memory mapped SPU based registers to be examined or modified. The contents of the register will be displayed. Entering a [return] will cause the next register to be displayed. Entering a hex value followed by a return
will change the contents of the register if it is a writeable register. Entering a ^ will cause the previous register to be displayed . Entering a **q** will exit this mode. The valid register names are:

cdr	console data register
rsr	remote control/status
rdr	remote data register
sass	sasi status register
sasd	sasi data register
tcr	timer control register
tdr	timer data register
isr	interrupt status register
isrm	interrupt status register (most significant word)
isrl	interrupt status register (least significant word)
ier	interrupt enable register
ierm	interrupt enable register (most significant word)
ierl	interrupt enable register (least significant word)
cop	cop register
pis	pbus interrupt channel
dsr	diagnostic switch register
ber	bus error register
per	parity enable register
rhr	run/halt register
sdr	scan data register
ecr	event counter register
dcr	diagnostic control register

cmr	clock mask register
cgr	clock gating register
esr	error status register
atd	analog to digital converter register
emr	enviromental monitor register
cpr	control panel register
srr	system reset register
mcr	margin control register

rd regname This command will display the bits in the named register. Each bit position will be labeled with the function of the bit. Each bit can be individualy modified by moving the cursor over the bit and entering a 0 or a 1. The cursor will move to the right (towards the lsb) when a new bit value is entered or when the space bar is depressed. The cursor will move to the left when a DEL or BS key is pressed. The input is terminated when a RETURN , **q** , ^ or = is entered. A RETURN will update the register and then display the next register. A **q** will exit this mode. Entering a ^ will update the register with the modified value then display the previous register. Entering a = will update the register with the modified value then redisplay the same register.

q[uit] This command will exit **spuutil.**

? This command will print out a list of valid commands.

NAME

syshalt – immediately halt the C1 computer system

SYNOPSIS

syshalt

DESCRIPTION

Syshalt performs an ungraceful halt of the C1 CPU by clearing the run bit in the run/halt register (RHR) and clearing all bits in the clock mask register (CMR). This command should only be used when all other means of gracefully halting the CPU have failed..

NAME

sysreset – reset the C1 computer system

SYNOPSIS

sysreset [**ccus**] [**jp**] [**mem**]

DESCRIPTION

Sysreset is used to reset the C1 computer system. If no subsystem names are specified, then all subsystems are reset. Otherwise, only the specified subsystems are reset. Subsystem names are as follows:

ccus	Channel control units
jp	Central processor
mem	Main memory

Sysreset disables hard and soft error interrupts by clearing the appropriate bits in the interrupt enable register (IER) and then sets all bits of the clock gating register (CGR). The clock mask register (CMR) and the system reset register (SRR) are set in accordance with the subsystems to be reset. Reset is asserted, a burst of 16 clocks is generated, and the SRR is cleared. The internal state of the boards comprising the specified susbsystems is initialized via the scan rings. If **jp** is specified, then virtual addressing and all caches are disabled. **Sysreset** leaves the CMR cleared.

FILES

/mnt/bin/initall

SEE ALSO

initall(1D)

NAME

vioputil – viop register/memory utility

SYNOPSIS

vioputil [VIOP number 3..7]

DESCRIPTION

Vioputil is a utility to display and modify VIOP memory locations. When the modify mode is entered the displayed value can be changed by entering the new hex value and hitting RETURN. If no value is entered the next memory location will be displayed. A **q** will terminate the modify and will return to the prompt

COMMANDS

Commands are of the general form:

command parameters

All address values are in hex. A **q** can be used to terminate any command.

m a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents.The locations are displayed/modified one byte at a time.

mb a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one byte at a time.

mw a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one word at a time.

ml a1 [a2] The contents of memory locations a1 through a2 are displayed. If a2 is not entered then location a1 is displayed allowing modification of its contents. The locations are displayed/modified one longword at a time.

f a1 [a2] value The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fb a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this occurs on a byte basis. If a2 is not specified then only location a1 is changed.

fw a1 [a2] value
The contents of memory locations a1 through a2 are filled with value, this

occurs on a word basis. If a2 is not specified then only location a1 is changed.

fl a1 [a2] value The contents of memory locations a1 through a2 are filled with value. This occurs on a longword basis. If a2 is not specified then only location a1 is changed.

M a1 disp This command will cause the memory modify mode to be entered. Memory will be 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 This command will copy a block of memory starting at a1 through a2 and place at a3.

rm regname This command will allow the contents of memory mapped VIOP based registers to be examined or modified. The contents of the register will be displayed. Entering a [return] will cause the next register to be displayed. Entering a hex value followed by a return
will change the contents of the register if it is a writeable register. Entering a ^ will cause the previous register to be displayed . Entering a **q** will exit this mode. The valid register names are:

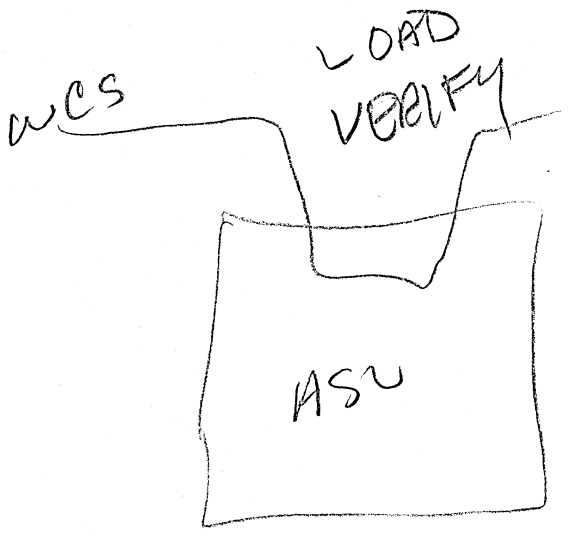
ier	interrupt enable register
ierm	interrupt enable register (most significant word)
ierl	interrupt enable register (least significant word)
isr	interrupt status register
isrm	interrupt status register (most significant word)
isrl	interrupt status register (least significant word)
mcr	memory control register
pis	pbus interrupt channel
ber	bus error log register
pber	parity/bus error address
pblgm	pbus log register (most significant word)
pblgl	pbus log register (least significant word)
pic	pbus interrupt channel number
clg	cache log
earm	error address register (most significant word)
earl	error address register (least significant word)
trr	test results register
auxtrr	auxilliary test results register
mbdr	multibus diagnostic register
misc	miscellaneous diagnostic register
pstat	parity status register
slotid	slot id register

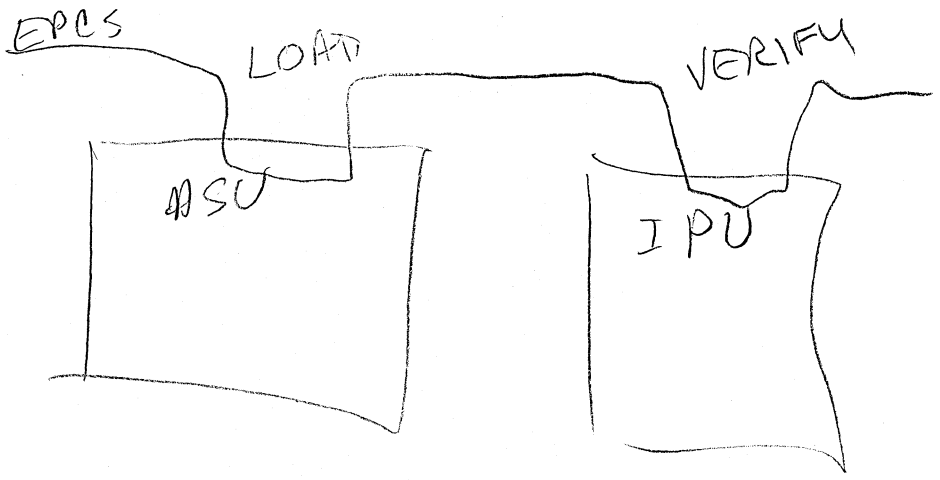
rd regname This command will display the bits in the named register. Each bit position will be labeled with the function of the bit. Each bit can be individualy modified by moving the cursor over the bit and entering a 0 or a 1. The cursor will move to the right (towards the lsb) when a new bit value is entered or when the space

bar is depressed. The cursor will move to the left when a DEL or BS key is pressed. The input is terminated when a RETURN , **q** , ^ or = is entered. A RETURN will update the register and then display the next register. A **q** will exit this mode. Entering a ^ will update the register with the modified value then display the previous register. Entering a = will update the register with the modified value then redisplay the same register.

q[uit] This command will exit **vioputil.**

? This command will print out a list of valid commands.


WCS
LOAD
VERIFY
ASU


EPCS
LOAD
VERIFY
ASU
IPU

NAME

wcs – load the C1 writable control store

SYNOPSIS

wcs [-p] [-l] [-v] file

DESCRIPTION

Wcs loads the C1 writable control store (WCS). **File** specifies the name of the WCS image file to be loaded. The file name extension is determined by the ASU ring revision. Scan ring revisions of less than 100 are designated for the previous ASU and have a file extension of *.wcs*, but only the root name of the file should be specified on the **wcs** command line. Ring revisions of greater than 100 are for the new ASU (FASU) and have a file extension of *b.out* format with the WCS image located in the text segment and a 32 bit checksum located in the data segment. The file may be generated directly from the HOST *microasm .h* output file via the HOST utility **wcsgen(1)**.

When executed, **wcs** halts the central processor, loads the specified file into the Service Processor Unit (SPU) memory, performs a checksum verification on the file contents, and then loads the file image from SPU memory to the WCS. After loading is complete, **wcs** reads the content of the WCS and compares it with the file image in SPU memory. If the WCS contents fail to compare, a maximum of five errors are logged before **wcs** is aborted. If an error is detected, **wcs** returns an error code. Otherwise, **wcs** returns 0.

The following options are recognized. If no options are specified, the default is **-lv.** In all cases the checksum verification is performed prior to any other action.

-p Load WCS image into SPU memory and prompt user with menu for subsequent action. Possible responses are **l** for load, **v** for verify, or **q** for quit.

-l Load only. Do not verify the contents of the WCS after loading.

-v Verify only. Compare the current contents of the WCS to the file specified.

FILES

/mnt/bin/initall

SEE ALSO

initall(1D)
HOST: **wcsgen(1)**, **b.out(5)**

NAME

x – hexadecimal/decimal calculator

SYNOPSIS

x [**expression**]

DESCRIPTION

X evaluates an arithmetic expression using 32 bit integer arithmetic. The expression may include hexadecimal numbers, decimal numbers, octal numbers, ( )'s, and the operations - (unary negation), ~ (unary inversion), *, /, % (remainder), +, and - . Unary operations are given highest precedence; multiplication and division next; addition and subtraction have lowest precedence. ( )'s may be used to override precedence. Hexadecimal numbers are specified with leading zeros (e.g. 017 = 17 hex = 23 decimal) or with a leading **x**. Octal numbers are specified with a leading **o** (letter o).

If no arguments are given, **x** prompts for expressions, one per line. To exit, respond with **q** or <CR>.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5
Diagnostic File Formats

5.1 Overview

This chapter contains detailed explanations of all pertinent diagnostic file formats.

THIS PAGE INTENTIONALLY LEFT BLANK

NAME

DB_cop – system board configuration database file

DESCRIPTION

The file *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 (i.e., asu, atu, etc.), and optionally, the name(s) of the card cage slot(s) where the board may reside (i.e., ccu7-3, etc.). If the slot name is not specified, the board type is used for slot identification. In addition, one or more lines specifying an assembly revision range with the corresponding diagnostic scan ring revision are listed. 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. Three line formats are valid in *DB_cop*. Lines that begins with a # are assumed to be comments. An example format follows:

```
# Example Cop Data Base File
#
# Entry format:
#
# p_num    b_type    s_name
#          asm_rev   rng_rev
#
# where:
#   p_num   = board part number
#           (must NOT be preceded by white space,
#            leading zeroes optional)
#   b_type  = type of board (asu, atu, etc.)
#   s_name  = optional slot name (ccu7-3, etc.)
#   asm_rev = assembly revision range (a, a-b, etc.)
#   rng_rev = scan ring revision number (1, 2, 3, etc.)
#
000129  asu
        a-b     1
        c-d     2
001129  asu
        a-zz    2
001134  asu
        a-zz    100
002125  atu
        a-zz    4
001132  hsp     ccu7-3
        a-zz    1
```

SEE ALSO

cop(1D)

NAME

DB_diskfmt – disk parameters for diagnostics

DESCRIPTION

The **DB_diskfmt** utility contains all essential disk drive parameters needed by the Multibus disk and controller diagnostic (*dev4100*), the Multibus disk formatter (*dev4110*), and the combined VMEbus disk controller and drive diagnostic and formatter (*dev5130*). Each line is either a comment (starts with a **#**) or defines a drive's parameters needed to properly perform I/O to the drive and for format of the drive. Below is an example of the complete file as released in Diagnostics Database V2.6.

```
# DB_diskfmt - file of disk parameters
#
# >>>> WARNING - DO NOT USE 'diskfmt' TO FORMAT! It is no longer compatible
# >>>>           with the CONVEX FORMAT! Instead, format MBUS-attached drives
# >>>>           with 'dev4110' and VME-attached drives with 'dev5130'.
#
# KEY FOR DRIVE NAMES (unformatted capacity is given in parentheses):
#
#  Name        Description          Name        Description
# DKD-001      Fujitsu Eagle (452MB) DKD-008,208  NEC 2363 (1080MB)
# DKD-002      CDC 9766 (300MB)      DKD-214      Hitachi DK514-38 (356MB)
# DKD-005,206  NEC 2352 (500MB)
#
#
#------- XYLOGICS 450/451 SMD CONTROLLER (MBUS) -------
#  a    b   c   d  e   f    g    h    i j k l m  n  o p
DKD-001 2  842 20 46 45 4800 28160 1 0 1  0  0  smd mfm y
DKD-002 0  823 19 32 31 5040 20160 1 0 1  0  0  smd mfm y
DKD-005 0  760 19 60 59 4832 36288 1 0 1  0  0  smd 2-7 y
DKD-008 1 1024 27 68 67 4816 40960 1 0 1  0  0  smd 2-7 y
#
#------- INTERPHASE 4201 ESDI CONTROLLER (VME) --------
#  a    b   c   d  e   f    g    h    i j k l m  n  o p
DKD-214 0  903 14 51 50 4736 30240 5 5 1  8  8 esdi 2-7 n
#
#------- INTERPHASE 4200 SMD CONTROLLER (VME) ---------
#  a    b   c   d  e   f    g    h    i j k l m  n  o p
DKD-206 0  760 19 60 59 4832 36288 5 4 1 12 12  smd 2-7 n
DKD-208 0 1024 27 68 67 4816 40960 5 6 1 16 12  smd 2-7 n
#
# LEGEND:
# a - drive name              Must be DKD-0XX for Multibus and DKD-2XX for
#                           VMEbus
# b - disk type               For Xylogics controller
# c - # of cylinders
# d - # of heads
# e - # of physical sectors     Number of actual sectors excluding runt
# f - # of logical sectors      Number of physical sectors (e) minus number of
#                           spares
# g - bits per sector         Number of bits between sector pulses
# h - bytes per track          Total number of unformatted bytes per track
# i - skew                   Sector offset from one head to the next
```

```
#                          Must be 1 when using Xylogics 450/451 cntlr
# j - # of relocation tracks    .5% of number of cylinders (c). Raise
#                          fractional part to next higher whole number.
#                          Ignored by dev4110 (Multibus formatter)
# k - interleave             sector separation between consecutive sectors
#                          Currently must be 1 for Xylogics 450/451 cntlr
# l - gap 1 size             Number of halfwords in gap before header
#                          (2 bytes per halfword)
#                          Ignored by dev4110 (Multibus formatter)
# m - gap 2 size             Number of halfwords in gap following header
#                          (2 bytes per halfword)
#                          Ignored by dev4110 (Multibus formatter)
# n - drive interface        Used to determine how to read manufacturer's
#                          defect map.  Currently, smd or esdi
#                          Ignored by dev4110 (Multibus formatter)
# o - data encoding scheme   Way data is encoded on the media. Used to
#                          select patterns for pattern test.
#                          Currently mfm, 2-7 or 1-7
# p - Are spares interleaved  For Xylogics, 'y'. For Interphase, 'n'.
```

Note that file contains details about each of the fields.

SEE ALSO

get_defects(1D)
disktab(5)

NAME

softlog – soft memory error log file for *errintd*

SYNOPSIS

/mnt/softlog

DESCRIPTION

This file contains a log of corrected main memory system single-bit errors (i.e., 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 (e.g., 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:

MAU device S/N bus bit stk first-fail last-fail address count

MAU
is the number of the memory board containing the faulted device.

device
is the coordinates of the faulted device, coded as follows:

CC-RRRS

where:

CC	=	failed device column number (e.g., L9)
RRR	=	failed device row number (numeric chars only)
S	=	MAU side (surface mount tech. boards only) ***

*** The side will only be present if the MAU is a 128-Mbyte Surface Mount Technology (SMT) type board. The side is reported as "F" for front or "B" for back. The "F" side is the same as the top side of a one-sided (non-SMT) board.

S/N
is the serial number of the MAU containing the above device.

bus
indicates from which bus (e.g., MBUS or PBUS) the request originated. This only applies to the last corrected error that occurred on this device.

bit
is 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 (i.e., a bit is stuck at the address under test). This only applies to the last corrected error that occurred on this device.

first-fail
is the date the first error on this device occurred.

last-fail
is the date the most recent error on this device occurred.

address
is 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
is the total number of single-bit errors (from this device) that have been corrected.

The **softlog** file may be examined from CONVEX UNIX by entering the following command:

/usr/convex/spu -r /mnt/softlog

SEE ALSO
errintd(1d)
mm_sniff(1d)

Appendix A
Reporting Problems

A.1 Overview

The *contact* utility is the recommended way to report minor hardware deficiencies and technical documentation problems to the Technical Assistance Center (TAC). This utility is an interactive tool that prompts the user for the information to properly file a problem report.

> NOTE
>
> The *contact* utility is not intended for requesting customer service for hardware failures. To restore your CONVEX equipment to operational status, faster service can be obtained by directly telephoning the TAC (refer to "Technical Assistance" in the preface).

To use the *contact* utility, there must be a phone connection to the TAC. A UNIX-to-UNIX Communication Protocols (UUCP) allows communication between UNIX systems by either dial-in or hard-wired communication lines. For more information, refer to *uucp(1)* or to the *info(1)* entry in the UNIX man pages.

The name and version number of the product involved is required. Use the *vers* command to ascertain the program or utility name and version. The syntax for the command is **vers** *filename*, where *filename* is the full pathname of the program. If the full pathname of the program is not known, enter **which** *program*. For more information, refer to the *vers(1)* and *which(1)* entries in the UNIX man pages.

A.2 Information Required to Report a Problem

The *contact* utility requires the following information:

1. The customer name, title, phone number, and corporate name
2. The hardware nomenclature, part number, and revision level, or the technical manual name, document number, and version

> NOTE
>
> Use *vers* and *which* to identify product name and version.

3. A short (one line) summary of the problem
4. The more information provided, the more quickly the problem can be isolated and solved. At a minimum, include a detailed description of the problem (including page references, if applicable), the source code, and a stack backtrace whenever possible.

> **NOTE**
>
> See the *adb(1)* or *csd(1)* man pages for information on obtaining stack backtraces.

5. The priority of the problem, selected from a list of six levels
6. Instructions on how to reproduce the problem, including the command syntax used, any flags invoked, or anything else attempted to make the program run
7. Any other comments about the problem or files to be submitted

The *contact* user has a chance to review and edit the report prior to submitting it. If the user decides to delay submitting the report, the session can be aborted. The report is automatically saved in the user's top-level directory in a file named *dead.report*.

See the following figure for a sample *contact* session. User input is in bold lettering, and the system response is in monospace type.

Figure A–1, Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University r
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed."  Happily, this is not true! (RETURN)
> (CTRL-D)

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 (RETURN)

Problem report submitted.
%
```

THIS PAGE INTENTIONALLY LEFT BLANK

Index

Symbol

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

THIS PAGE INTENTIONALLY LEFT BLANK

CONVEX Diagnostic Utilities Manual
(C1, C120)
Document No. 760-001050-201, Second Edition

Reader's Forum

You are invited to submit comments concerning the clarity and service of this manual. Constructive critical comments are most welcome, and will help us continue in our efforts to generate quality customer documentation. Please list the document page number with your questions and comments.

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____Title_____

Company_____Date_____

Address and Phone No._____

FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

Location	Phone Number
In Texas	(214)952-0200
Other continental locations	1(800)952-0379
Outside continental US	Contact local CONVEX office

Direct mail orders to:

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

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)